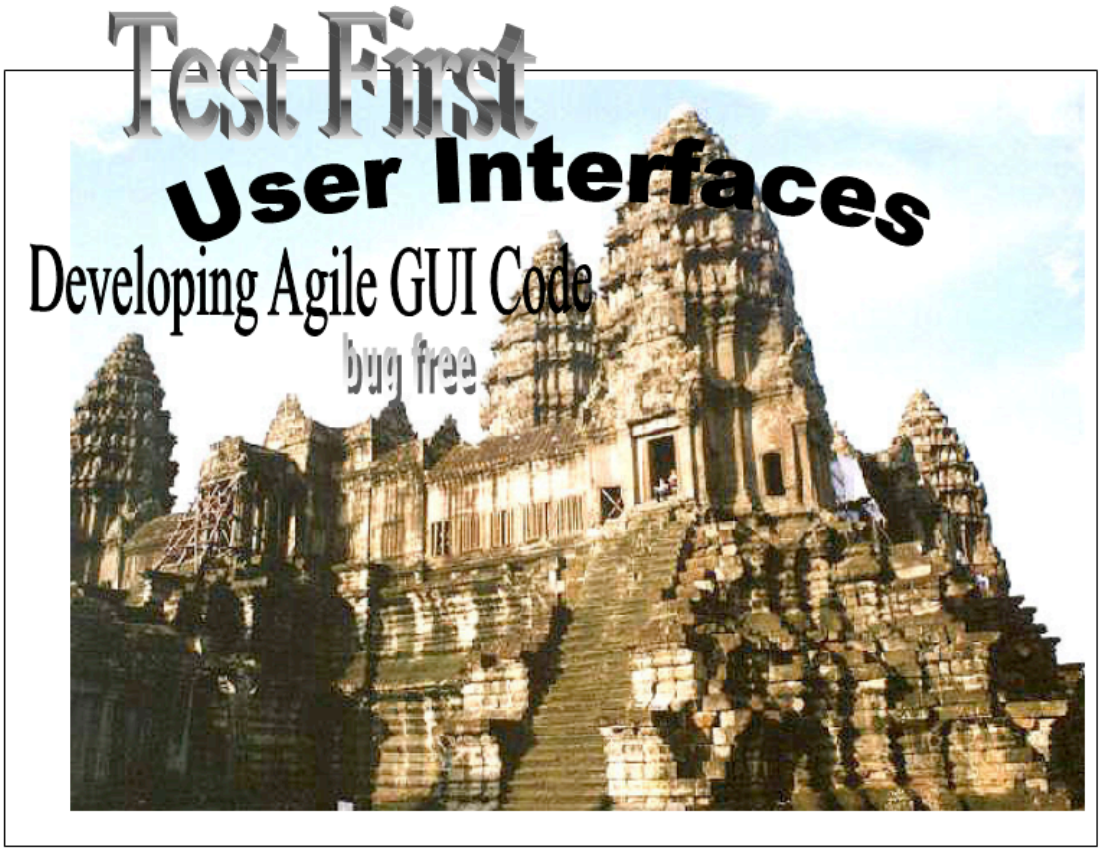

Test First
User Interfaces
Developing Agile GUI Code
bug free

Dedicated to Ashley & Iris

(in recognition for continually reminding me that real life is much more important than writing books;)

The cover features Angkor Wat, the largest temple in the remains of the capital of the Khmer civilization. Built from 879 to 1191 CE in Cambodia.

Contents

Foreward

Oh, people can come up with statistics to prove anything, Kent. 14% of people know that.

—Homer Simpson

Acknowledgements

I would like to thank Kent Beck, Peter Merel, Ken Happel, Ron Jeffries, Tom Copeland, John Sarkela, Sunitha Dangeti, Corey Goldberg, Unnsse Khan, Hidetoshi Nagai, My Hood, My Folks, Samuel Falvo, Lisa Crispin, Dan Twedt, Beth Crespi, all the Internet forums, Chris Hanson, Jeff Grigg, XP SoCal, and XP San Diego. My long-suffering reviewers were Ben Kovitz, Siemel Naran, Kay Pentecost, Tom Poppendieck, Stan Rifkin, and Dossy Shiobara.

Incomparable technical gratitude extends to everyone cited in the text (and anyone overlooked), for writing a good resource or creating an excellent software module.

Introduction

When developers use tests to design code, they enable changes and prevent bugs. Some libraries and environments (& cultures) make automated testing very hard. Graphical User Interfaces are the worst. All other systems submit to program control; GUIs have a side only users see or touch. Their visual appearances and input behaviors resist tests, and they are too easy to accidentally change. These side effects complicate writing tests that force predictable changes in GUI appearance and behavior. Predictability leads to Agility.

A naïve attempt to write a test on a GUI might lead to a window popping up, blocking the test run. This window tempts programmers to visually inspect it, click on it, drive it, debug it, and inspect the results. That way leads to a madness familiar among GUI programmers.

Seven years ago, my abilities reached limits. I knew only how to design, code, and debug (in that order), not how to sustain those activities as projects scaled. The more code a project ran, the higher the bug rate for each change, and the lower our visibility to managers and experts. At that time, Smalltalk programmers, on the distant opposite side of the industry, kept secret how to go farther. They conveyed their ideal in a new set of jargon, and no one knew how to explain what they were doing (except via Pair Programming).

These books sprung their secret:

- *Test Driven Development: By Example*, and
- *eXtreme Programming eXplained: Embrace Change* by Kent Beck
- *Refactoring: Improving the Design of Existing Code* by Martin Fowler

The secret was to test, code, and design, in that order.

When those authors put words to experiences that felt subjective, they learned these words were not grandiose pep talks; they were very specific guidelines to programmer behavior. Configure your project so when it does X_i, you instantly do Y_i. Learn those stimuli and responses, and repeat them over and over again. The right set of simple rules can generate very sophisticated behaviors.

Those books teach which Xs indicate what Ys, and how to minimize any latency between feedback and response. Testing, refactoring, and teamwork make development rapid and predictable, but many libraries and systems interfere with automated feedback, flexibility, and

sharing. This book introduces Agility in GUI terms, and reveals how test-first can develop user interfaces.

The book has three parts:

- Part I: One Button Testing, 6—how to develop like this
- Part II: Case Studies, 57—what to do in specific situations
- Part III: Explications, 419—why developing like this works

Read Part I, and imagine applying its concepts to your project. Each concept in Part I links to complete examples in Part II. Follow a link to see the concept working in a real, complete project. Download the source of this project, from http://zeroplayer.com/tfui/TFUI_source.zip, and follow along with the code changes.

Part II illustrates many practical techniques and patterns, supporting and extending our TFUI Principles. Part III explains Agile practices in general, with links back to their technical implementations.

Who Should Read this Book

Any engineer, using any platform and methodology, interested in a GUI's success.

Teams creating modest or elaborate windows will learn a new development strategy. This book's strategy upgrades your interactions with your colleagues, so they should also learn the new jargon and lifecycle. The strategy supports any methodology; this book illustrates the strategy working within Test-Driven Development and Extreme Programming.

This book pre-requires no other methodology book. Experience programming GUIs, using Object Oriented languages, is preferred.

This book is not a thin manifesto, but a hands-on experience. Begin to read by selecting one of several tracks through the material.

Developers

This book delivers a light, scalable strategy for "test-infecting" all kinds of GUIs. Part I reveals the strategy in platform-neutral terms. Sample code for each technique in the strategy appears in Part II: Case Studies. Each small project presents simple solutions for very hard GUI problems. Part III links the strategy to the "Agile" methodologies, and extends each Case Study with suggested Exercises.

If you are "the GUI guy", this book will boost your velocity, and open new potentials. If you are *not* "the GUI guy", this book will increase your reach into that realm.

You may need a technique whose code examples might not use your platform. All GUI platforms have common abilities, and all unit test frameworks in the SUnit, JUnit, NUnit family have common facilities. This book's Case Studies create brief but powerful applications following the same strategy. It will work for you, no matter what your platform. The Case Studies use domains and environments as diverse as embedded software and the World Wide Web.

To **get started quickly**, read this entire Introduction, skim Part I, and read Chapter 2: *The TFUI Principles*. Then read either the most familiar Case Study, or the *least* familiar one. To "play along", get the online source, decompress the archive matching the section that challenges you, install its few dependencies, and start testing and refactoring. Read a Case Study's development session and enter each edit, or read Chapter 17: *Exercises* to think of a

new feature to add. Then return to your own GUI project, to write tests with the extra features this book recommends.

If you use a published GUI test rig (such as Abbot, AutoRunner, Cactus, Canoe, Jemmy, JfcUnit, HtmlUnit, HttpUnit, NUnitASP, Rational Visual Test, Samie, Selenium, SilkTest, StrutsTestCase, TestPartner, Watir, WinRunner, etc.), lead it to support the techniques recommended here. Some come very close, so this book need not dwell on all of them. They cover gaps in GUI libraries; this book focuses on the gaps themselves. Some GUI test rigs provide new gaps.

If you have no pre-existing GUI test rig, follow this book's techniques to rapidly create a minimal and proactive one tuned to your project's topology.

If **Test-First Programming** (TFP) already drives your GUI, skip directly to the Principles *Temporary Visual Inspection* (page 18) and *Temporary Interactive Test* (page 18).

If a Case Study uses a GUI platform that you are learning, this book supplements its tutorials and references. It recommends you ask questions that they might not answer, so you will also become familiar with your platform's online communities!

Journey through this book with or without the company of any of the fine Agile books that cover the general theories and/or your specific platform. If you don't have one (yet), Chapter 16: *Test-First Programming* explains the TFP lifecycle, and Chapter 2: *The TFUI Principles* adds GUI enhancements. Chapter 5: *SVG Canvas* and Chapter 6: Family Tree illustrate test-first and refactoring in detail, producing a brief yet powerful project. These Chapters use a simple and portable GUI platform that rapidly enables these techniques. If you play along, you will soon forget that you are testing-first a GUI.

Your own project, platform, and test rig might not enable these techniques so rapidly. Hence the rest of the book.

Students

Those who have not yet read any other methodology book should read Part III first. It introduces the "Agile" and design topics that other Parts assume you know. As that Part raises question about general software engineering, follow its citations out to the relevant books. These are listed in the Bibliography on page 474.

Testers

An Agile project begins when testers convert business requirements into technical specifications that can be tested. This book demonstrates one of several Acceptance Test Frameworks that host these specifications as literate tables of data, and execute them against live code.

This book teaches your developers to use Test-First Programming to implement the features that then pass those acceptance tests. TFP produces a clean design and robust code, with many low-level test cases as a by-product. TFP does not produce industrial-strength tests (and bookstores should not shelve this or other Test-Driven Development books with the "Testing" or "Quality Assurance" books). Agile projects need testers to add the real tests that attack code with twisted scenarios to find its boundaries, and collect metrics to rate its health.

You will have time to do all that, because TFP rapidly produces decoupled code that resists bugs. You will find new tests very easy to add, and defects very easy to characterize. Long hunts for the sources of bugs will become very unlikely.

Read this entire Introduction, Chapter 13: *Agility*, Chapter 16: *Test-First Programming*, and Part I. Then skim Part II to find "attacks" suitable to your situation.

Usability Experts

No automated test can judge usability. This book presents only a trace of GUI design theory. Yet this book will change how you specify, design, and review GUIs.

This book helps teams keep GUIs flexible, robust, and visible, permitting aggressive upgrades, automated tests of usability details, and multiple "skins" for diverse user populations.

Read Chapter 4: *The Development Lifecycle with GUIs*, and the second half of Chapter 9: Broadband Feedback.

Leaders

Without Agile practices, software engineering projects occupy a spectrum, from excess paperwork to undisciplined hacking. GUI implementation trends toward the hacking end, lead by vendors' tools that make GUI prototyping appear as easy as painting. Their tools irresistibly link painting and coding to code-generating wizards and full-featured debuggers. They neglect hooks to enable testing. These biases resemble old-fashioned manufacturing methodologies that planned to over-produce hardware parts, then measured them all and threw away the defects. Both speculative planning and endless hacking risk extra rework and debugging, as fixing old bugs causes new bugs.

Agile projects rise above that spectrum. We apply discipline to the good parts of planning *and* hacking. We carefully plan to write many tests, until new features are as easy as hacking—without the low design quality, high bug rate, and inevitable decay. Agile projects become useful early, add value incrementally, and maintain their value.

Most "Agile" books compare Agility to methodologies that analyze requirements and plan designs in big batches before implementing them. This one compares it to our distinguished heritage of hacking and debugging GUIs.

Use this book along with any of the fine Agile resources on the market—in print and in person—to adjust your teams' behaviors, and upgrade their results. Read this entire Introduction, Part III, and Chapter 4. Help your crew experiment and play with their GUI libraries, to introduce Chapter 2's technical recommendations. This brief investment will reinforce your new process, and pay your project back many times. An ounce of prevention is always better than a pound of cure.

This book recommends extra research into corners of GUI architecture that might be a little dark. It shows programmers how to convert that research into extra code that makes tests easy to write. The goal is rapid, scalable, and sustainable development, without more excessive research. Expect to see this progress; it will appear quite different from the usual GUI hack-n-slash.

Style

Read this—it's not the usual "yadda yadda yadda".

- The word Pattern as a Proper Noun cites either the book *Design Patterns*, by Gamma, Johnson, Helm, & Vlissides, or *Smalltalk Best Practice Patterns* by Kent Beck. Common pattern-like things without a "Pattern Language" deserve only a lower-case "pattern".

- Likewise, Refactor as a Proper Noun cites the book *Refactoring*. It means changing a program's structure while leaving its behavior the same.

- An “AntiPattern” is a common software engineer behavior that inhibits productivity. Our industry depends on far too many AntiPatterns for every use of the word “AntiPattern” to cite the book *AntiPatterns: refactoring software, architectures, and projects in crisis* by Brown, Malveau, McCormick & Mowbray.

- A **Bold Proper Noun** heralds the first use of a term from the Glossary.

- “Quoted Proper Nouns” are typically invitations to search the Internet and learn more about a peripheral topic.

- This borrowed icon  besets links from the context to the Agile concepts.

- `monospaced` words appear inside computers (but library and module names appear as Proper Nouns, and documenting comments appear in a normal roman font).

- **`bold`** `monospaced` words are source code changes. The `non-bold` parts are often cloned code, then the **`bold`** parts are modifications.

- *`italic`* `monospaced fileglobs` *`?*`* are placeholders for reader-supplied values inside source code.

- Pronounce empty parentheses as “the function” or “the method”. So “`putSvgIntoCanvas()`” becomes “the function put S, V, G into canvas”. This notation may elide arguments.

- <Key Caps> are keyboard key names.

Part I: One Button Testing

Test-first is rapidly becoming the leading software implementation technique. GUIs, by nature, resist any kind of test, and strongly resist test-first. This book proposes that a lean, right-sized test rig, grown with its target GUI, and using the same tools, can efficiently streamline development.

- **Chapter 1:** ***The GUI Problem***—Something about GUIs makes test-first very hard. We must isolate the problem before dealing with it.
- **Chapter 2:** ***The TFUI Principles***—A strategy to put any GUI under test with minimal overhead.
- **Chapter 2:** ***GUI Architectures***—Applying the strategy to each of the three kinds of GUIs, leading to specific development patterns.
- **Chapter 4:** ***The Development Lifecycle with GUIs***—placing the strategy into the larger context of teams and projects

Chapter 1: *The GUI Problem*

The reasons to write test cases before writing production code are simple yet subtle. The most compelling reason is to keep software easy to change over time. You might know how to write a new function, now, without its test, but a long time from now you might not realize how a change to another module could break this function. Nobody has perfect memory, and no team has perfect communication. Invest your understanding of your new function, now, into test cases that will raise an alarm if something breaks later. Run all the tests all the time. These simple rules boost development speed, while reducing the odds of debugging.

A recurring theme of this book is test cases must grow cheap and easy to write. When we test new code, as it grows, natural laziness provides simple and cheap tests. Each test case wants to test only one small behavior, so the tested function must have simple inputs and outputs. The test case should not construct every object in your program just to call that function. So that function will grow to decouple from other influences.

The most subtle and powerful effect of test-first is "emergent design". Tests help your classes decouple from each other, when each test case shows a class performing alone without excess help from other classes. Decoupled classes are easy to change without affecting other classes. Designs that frequently change, under test, become very robust and flexible. This effect is easy to do and hard to talk about. It's the main reason we take such small steps between each test run. Future chapters will illustrate some counterintuitive decisions, and will explain them with somewhat terse jargon.

This chapter uses a question and answer format to introduce that jargon. This early in the book, only thin metaphors and incomplete rationales can explain how to leverage testing. We must briefly reveal why test-first for GUIs is hard but important, and how programs with GUIs can improve their architecture to avoid complex techniques. The devils are in the details, and the next chapters will exorcize those.

What is Test-First Programming?

The Chapter of the same name (on page 443) illustrates the complete lifecycle.

Write a failing test case, write code to pass it, refactor the code to improve its design, and repeat for each tiny code ability. After the fewest possible edits—say ten at the most—run all the tests and predict their results. This technique makes the odds of bugs and the odds of excess and difficult refactoring very low.

Over time, TFP replaces long hours debugging with short minutes writing tests. Time spent debugging, without knowing the source of a bug, is time wasted. Time spent manually testing, not knowing if an innocent-looking change broke anything, is also time wasted. The trick is fixing both wastes at the same time.

When you debug, you feed a program inputs, run to a breakpoint, and examine an intermediate value. This is an experiment, with an hypothesis and a result. A test case is a permanent record of such an experiment. Each case runs a function and examines one of its values. When you run all the tests, you perform every experiment again. This is like instantly, manually debugging your entire program, many times in many ways, to reconfirm that each intermediate value is still correct.

Without tests, if you change code very rapidly—adding features or improving its design—you will create many kinds of bugs, some easy to find and some hard. With tests, you can change code rapidly while constantly reducing the chances of bugs. The more test cases, the more ambitiously you can change code. So change turns from a scary thing into the most useful tool in software engineering.

What's a Test Case?

Tests are code outside your application, written in the same language, and built by the same build scripts. Those should compile everything, run all the tests, and report their results.

A test case is one unit of testing. It assembles target objects, activates their test methods, and asserts the results were correct. Here's a test case, from an arbitrary project:

```
TEST_(FrameSuite, GetTitle)
{
    CHECK_NOT_EQUAL(NULL, pFrame);
    CHECK_EQUAL("Plot Utility", pFrame->GetTitle());
}
```

The test's target is `pFrame`, a member of `FrameSuite`. That class constructed the `pFrame` in a `setUp()` fixture (not shown), before running this case. The lines starting with `CHECK_` are assertions. When they fail, the editor or test script reports the failing conditions.

That case requires the production code to have a `Frame::GetTitle()` method.

The `TEST_()` macro is a fixture that automatically registers its case with the global list of cases to run. That prevents the need to call extra registration methods.

Chapter 7: *NanoCppUnit*, on page 161, illustrates this ultra-light test rig.

How Do Tests Make Development Faster?

When automated tests fail, you have many more options than when manual testing exposes a bug. Because tests document and cohere with their tested code, their failures often clearly indicate the problem. You briefly fix it and keep going.

Testing is (slightly) more important than designing or adding features.

Keeping code easy to change is more important than searching for the "perfect" design that covers all possible features.

When you test before designing, the design's most important requirement, testability, is always satisfied.

You can rapidly change projects with tests—refactoring or adding features—by running the tests after every few edits. Use TFP to produce tests that fail:

- **early**—when you run them

- **loudly**—so you can't ignore the failure
- **expressively**—so you can easily see what went wrong
- **reversibly**—so you can back out of the situation.

Tests fail **early** if you run them as often as possible, and you strive to express every aspect of your project as a test. They fail **loudly** when you bond them to your editor and environment to invoke a hardware breakpoint at the failing assertion. And they fail **expressively** when they report the failure conditions, and when each test case is very close to its tested code.

Can Tests Catch Every Bug?

Some folks ask why bother to write tests if they can't prove a program is bug-free. It's true; the tests for a complex program must run in geological time to exhaustively prove every combination of situations. That's no excuse not to test.

Use tests to prevent bugs. That frees up your schedule, and makes your code's condition highly visible, so you can easily catch the remaining few. Developers focus on tests that are easy to write. But…

Our list does not require tests to fail "**accurately**"! Most tests are accurate, of course. This book discusses ways to make all test failures relevant, without working too hard to make the tests perfect. No test can prove the absence of bugs. The next best thing is cheap tests that fail *too often*, rather than too infrequently. This book explores such **Hyperactive Tests**. They can force code into its most testable configuration, leading to very high coverage.

The Agile software development techniques, such as Extreme Programming, leverage unit tests and other development procedures that fail early, loudly, expressively, reversibly, and often accurately. That's why Agility is success-oriented; errors and mistakes get the earliest possible opportunities to attract attention leading to a fix.

What's the Best Way to Fix a Failing Test?

One myth of Agile development is it forbids using a debugger. In the ideal situation, you are *allowed* to use the debugger, but you are not *motivated* to. Test cases make an excellent platform for debugging, to learn about legacy code or review new routines. Convert such learning into new test cases as soon as possible.

When a test fails and you don't want to fix the problem, for whatever reason, tests give you another, very powerful option. The TFP cycle is **reversible**, so your editor's Undo system can always revert your code to the last state where all tests passed.

That state must be very recent, because you should not make more than 1~10 edits between passing all tests. After undoing, make even fewer edits between tests runs.

Why 1 to 10 Edits?

In your primary codebase, between test runs, you should only perform so few edits that they all fit in your short-term memory. You could manually reverse your changes, back to the last passing state, if you wanted to. Frequent testing positively reinforces and rewards your mental model of the code's situation.

If you can't think of a small edit, and need to research all the possibilities (see the middle of Chapter 11: *Fractal Life Engine* for this situation), copy your code out to a scratch project, and then party. When you learn what to do, return to the primary codebase, and exploit your research to perform small edits.

Why Test FIRST?

Prevention is better than a cure. To replace debugging with a more robust implementation technique, write tests before the risk of bugs arises. When you write a test and predict failure, you test the tests at the most efficient time. Inspect your assertions' outputs to make certain the test failed for the correct reason, and that the assertions' outputs are useful. Writing code to pass such a test stakes out the territory where it can't regress.

Test-first is the most rapid and aggressive way to develop code that fails early, loudly, expressively, and reversibly. The code to pass a test should be simple, and the subsequent refactor to improve design should promote elegance.

Simple code resists bugs. Tested code is easy to simplify. Simple code is decoupled, and easy to test. This cycle squeezes waste out of your process. TFP searches a path of simple code for a clean design that satisfies all committed requirements.

Test-first for GUIs is hard as the temptation arises to "just look at" the GUI to see if new code works. If a test infrastructure can force *predictable* visual changes, it can lead advanced GUI modules to the same level of bug resistance and communication that other modules enjoy.

How Do Tests Help Requirements Gathering?

This chapter makes adding new features sound too easy. In a business environment, we must learn to use our powers for good and not evil. The goal of software engineering is to efficiently locate that 20% of features, or less, which will return 80% of the value, or more, for our users. Don't write too many features, just to log billable hours.

Tests help you frequently demo, review, and deploy your new features. A passing test batch should imply very high confidence that your project is ready for immediate delivery. Any question or impediment regarding this status should be answered with more tests.

Your customers should profit from your features as soon as possible, so they can learn what features to request next. This extends control over your project to those who need your features. Some software projects make this feedback loop as tight and comprehensive as possible using a simple trick. They build a cheap and flexible user interface for their test rig, so customer representatives can easily write new cases. This book examines such rigs in narratives and in sample code.

How Do Tests Sustain Growth?

Small projects are as easy to debug as to test-first. The goal is small projects that grow large with customer requests.

As a project grows without tests, it resists changes. To add a new and unforseen feature, you should reconcile its design with the rest of your program. This change should ripple through your system until the design is as clean as if you had predicted that feature. If you change too little, you lower the design quality. If you change too much, without tests, you introduce bugs. Either way, change without testing gets harder and harder over time.

A healthy test rig is an investment in a program's future. It allows refactors that merge the code of new features with old ones. It prevents bugs while these refactors force code to grow more flexible and more bug-resistant.

When a project is out of control, the effort required to add each new feature can look like this:

The solid line is the cost of each feature. Even when features are the same size (when each requires the same number of code changes), the effort line can vary widely. But varying effort, alone, won't make a project hard to control.

Each dotted bar represents the optimistic and pessimistic estimates for those features. If you can't estimate, you can't schedule a project.

In an out-of-control project, you never know which feature will be the one that causes radical design changes or long bug hunts. So all our estimates include "float"; very wide spans between optimism and pessimism. Sometimes a feature misses its window, and turns out harder than expected. Some features turn out easy, and waste their float times. Either way, useless estimates disturb our schedule.

Over time, the average effort trends upward. So the longer an uncontrolled project goes, the more risk it causes. The goal is projects that remain in control, like this:

The time and effort to code each feature is short, regular, and generally descending. More important, the estimates are narrow and increasingly accurate. Testing each change makes all changes easy and predictable.

That diagram assumes your project has invested in learning to make all tests easy to write. The sub-chapter Time vs. Effort, on page 53, compares the cost of a long project with poor tests, to the up-front cost of researching how to get difficult things under test.

What's So Special about GUIs?

All GUIs solve the same problem: Users dislike keeping a program's internal state in their heads. Projecting that state into pictures and animations leads to advanced and complex inputs. All GUIs collate elaborate input events from users, so all GUIs use an asynchronous event queue.

Asynchronous event queues are hard to test. But GUIs come with worse problems.

Why is TFP for GUIs Naturally Hard?

The target situation is this: We should be able to write new test cases that force predictable changes in a GUI's appearance and response. Passing such new cases should upgrade a GUI. Each passing test raises our confidence that our GUI's appearance and response has improved, so we only rarely look at or drive the GUI.

All other libraries submit to program control, making prediction easy. But GUIs have a side only users see or touch. We don't want to view the GUI after writing every test case, and we want to know that all tests constrain all our GUI features despite changes anywhere in a system.

Many tests call production code that sends a command to a GUI. Then the test queries the GUI to detect effects of that command. These tests might deceive when a GUI Toolkit supports asymmetric features. A program can change a graphic state, but can't reliably read back that state. A program can simulate an input, but only as a programming convenience, not an accurate simulation that behaves the same as a real user.

We conquer these problems by enabling test cases that temporarily present GUIs, under test, for inspection and interaction. This, in turn, helps us grow test fixtures to increase the odds that we can predict GUI changes.

Why is TFP for GUIs Artificially Hard?

GUI Toolkits have steep learning curves. This inspires toolkit vendors to compete by using their toolkits to build helper applications—wizards, form painters, debuggers, etc. A good environment can flatten a GUI Toolkit's learning curve, turning it from the hardest library in a project to the easiest.

So the **GUI Layer**, within a large project, may suffer neglect from much of a team's total experience. We will call the backend of a project the **Logic Layer**. When this layer is mysterious, complex, and valuable, it often receives more senior attention, and more careful development practices. When leaders think that GUI development is "easy", they might not pay so much attention to the growth of its design.

To put it indelicately, senior developers carefully write Logic Layers, and associate developers paint and debug GUIs. Chapter 14: *Sluggo at Work*, on page 429, illustrates this problem in disturbing detail.

GUI Toolkit vendors reinforce this culture. Their marketects claim, "Anyone can write simple data-entry forms with our system!" The fun starts as a team scales these simplistic forms up into rich and balanced applications.

A GUI's windows and buttons are easy to see, so they are easy to specify. A Logic Layer is hard to see, so developers might work harder to design its code structure. When developers work too hard, the code can fill up with excess design elements, making it resist change in unexpected ways. This AntiPattern, "Big Design Up Front", is the bane of much software engineering. GUIs typically have the opposite problem.

Much GUI development neglects design quality. Implementing new features by debugging muddy code until it just barely works leads to a familiar AntiPattern, "Code-and-Fix".

Both AntiPatterns can generate code that resists unexpected changes.

GUI Toolkit vendors reinforce Code-and-Fix by promoting elaborate and tempting wizards and debuggers, to help you generate code you don't understand, and then edit and debug it at the same time.

These interactive development environments often provide drag-and-drop interfaces to paint screens and wire up events. Such "design-mode interfaces" often obscure the techniques you need to bond controls to your test code.

Finding a way through these mazes, to write any test *first*, seems impossible.

The advanced features that vendors invest in their GUI Toolkits, to compete, interfere most with Agile development.

When test cases target your specific application more more powerfully than generic form painters, debuggers, and wizards, GUI development becomes rapid and safe from bugs and unwanted feature creep. If your GUI Toolkit vendor used Test-First Programming, and bundled their Toolkit with a suite of exemplary tests (or all of them), your life is simpler. The rest of us are not so lucky.

How to Avoid TFPing a GUI?

Agile projects avoid extra work using a peculiar but effective strategy, often called "You Aren't Gonna Need it". If you predict you may have a problem, such as a thick GUI that requires tests, you don't proactively solve the problem. Instead, you behave as if you don't have the problem, and develop normally, while seeking opportunities to force the problem to appear.

The classic example here is optimization. You should write clear expressive code, and don't prematurely obfuscate all of it on behalf of performance. (And you should use minor optimizations, such as `const &` in C++, that don't obfuscate.) If your product must perform efficiently, you should frequently time-test your entire application. Only when performance degrades do you speed up the code. This strategy collects hard data first, to make certain you solve only the right problem. Making clean code fast is easier than making fast code clean. You don't speculate, and make all the code faster; you right-size the effort, and only fix the code which your performance data shows is slowest. Notice that strategy implies you might store data tables in flat files until performance data indicate you need an expensive database.

Engineers seek ways to cause the problems they intend to solve. If they can't cause a problem, they are done.

To avoid using the TFUI Principles, use standard TFP to write a module that does everything your GUI does, but uses no GUI controls. We will call this the Representation Layer. Mike Feathers has given that technique the nickname "Humble Dialog Box", with this influential paper:

`http://www.objectmentor.com/resources/articles/TheHumbleDialogBox.pdf`

Anything a user can do to the GUI, a programmer can do to the module. If a user can click Submit, you have a `Submit()` method. If they can click on an element in a list box, you can pull a list of objects, and pass one into a corresponding function.

Anything the GUI can do to a user, this module does to programmers. If the GUI would disable that list box, then the programmer can detect the disability. If the GUI can refresh its display automatically when new data appear, the module can send a message using the Observer Pattern.

Now write the actual GUI, and make certain every event handler is as short as possible. It should do nothing but instantly delegate to the equivalent functionality in our wrapper module.

Other references might call our Representation Layer the "Logical User Interface", or the "Presentation Layer". A Representation Layer converts from one representation to another. Ours converts from the Logic Layer's format to the user's experience. We enforce our Representation Layer with a simple rule:

A Representation Layer may not use any identifier from the GUI Toolkit.

If your GUI Toolkit provides, say, `Label`, `Form`, and `Pushbutton` identifiers, your Representation Layer must not use any of them. This technique exposes your Representation Layer to the full benefit of logical design techniques without the encumbrance of GUI testing.

The Representation Layer converts database abstractions to user abstractions. Suppose the user can scroll a list box and select a name. The Logic Layer may or may not keep any names in any list. The Representation Layer re-arranges data to satisfy user needs.

So Why Bother TFPing a GUI?

Why do we insist on doing the impossible? No matter how thin you think your GUI Layer is, and how much logic you push out into the Representation Layer, the simple fact is bugs spontaneously generate in code without tests. TFP will illustrate this effect by pushing the defect rate down everywhere that tests go. Your program's testless parts will experience a higher defect rate, and will stick out like sore thumbs.

Many GUIs, including some of mine, will grow and live happy lives without any tests. We could easily follow the policy "Ain't broke don't fix it." We could decide not to write tests for some of our modules until we need to. Unlike a Logic Layer, faults in a GUI are trivial to find and fix. We could decide to "Capture Bugs with Tests", and only write tests after we detect bugs. So the Agile aphorism "You Aren't Gonna Need it" might lead us to defer labor, hoping to permanently avoid it.

That strategy has one major flaw, called "Activation Energy". If a module already has tests, new tests are very easy to write. The activation energy is low. However, modules without tests can make new ones very hard to write. (Hence entire books on the subject, such as *Working Effectively with Legacy Code* by Mike Feathers.) The activation energy is much higher.

The larger a module grows without tests, the higher its Activation Energy gets. When you receive defect reports, the energy required to "just fix them" is lower than the energy required to retrofit a test rig, capture the bugs with tests, and *then* kill them. So each time

you "just fix them", and put off that labor, defects grow more likely, the odds of putting off the labor gets higher, the risks of debugging go up, and the deferred labor gets harder.

Tests are the only aspect of Agile development that you *are* going to need.

Authoring

All new modules, including GUIs, must start with tests. That's the best way to learn where the trouble spots and nasty surprises are, to ensure that test suites grow, with their modules, and remain ready to assist in all advanced development.

However, not all GUI features need tests. Much of GUI development is authoring—painting controls, positioning them, coloring them, etc. If you don't intend to refactor the logic supporting trivial graphic details, author the graphic detail directly.

This book uses the verb "author" to mean, "Skip the tests, and just type this part in."

This book also explores advanced techniques to escalate the review of graphic details, in bulk, decoupled from their applications. That reduces the risks from faults in authored details.

But Aren't All GUIs Different?

How can the same strategy work on all of them?

All GUIs decouple the user from a program using an event queue. This routes input events, including commands to repaint the screen, into event handlers. Some handlers live in your GUI Toolkit, and some you declare in the GUI Layer and customize.

GUIs use three general systems. When the user invokes or uncovers a window, its **`Paint()` event** handler sends hardware commands to display the window on your monitor. A **control** is a unit of animation managed by your GUI Toolkit, so you generally don't need to worry about its `Paint()` event. A **script** is a batch of commands that generate or process controls, then submit them to an event queue.

Chapter 3: *GUI Architecture*, on page 26, explains these systems in relation to your application and tests. Each system yields to specific test attacks. Because they all use an event queue, the attacks are all different aspects of the same strategy.

Conclusion

Test-First Programming demands obedience from code. GUIs are persnickety, unruly, and ornery.

The first technique splits a project's GUI modules into a thin GUI Layer and a thick but GUI-free Representation Layer. That intermediates between the Logic and GUI Layers, exposing the maximum interface to aggressive tests.

But we can't completely avoid some effort. A thick Representation Layer does not absolve even the simplest GUI of the need for exploratory tests. A simple GUI might turn complex sooner than you can retrofit a test rig. This book describes how to write advanced GUI controls that supplement the ones provided by your GUI Toolkit.

Instead of neglecting GUI tests, or researching them forever, you need a strategy that forces discovery of GUI test issues, and right-sizes the amount of test infrastructure that your application's GUI will need.

Chapter 2: *The TFUI Principles*

The last chapter defined "test-first", where new tests force predictable improvements. This chapter describes the technical requirements that enforce test-first for GUIs, leading to a simple goal: You should be able to write new test cases that force your code to change a GUI's appearance or behavior, without looking at or driving the GUI to confirm each change.

Wizards, form painters, and debuggers are useful tools. GUI Toolkit vendors promote their products by ensuring you can generate, paint, and debug a GUI until its appearances and behaviors stabilize. This directly contradicts our goal of heads-down development, without displaying a GUI, so tests can upgrade its appearance.

How to Test-Infect a GUI

We must make sure our lowly test cases can compete with our vendors' offerings. We need to spend more time among our test cases than among our vendor's tools, or our target application's GUI. That requires giving test cases simple but useful GUIs of their own.

To get there efficiently, we will grow this GUI only by responding to real problems in our project. This chapter gives names to solutions for those problems, and shows how their solutions reinforce each other.

If you use an off-the-shelf GUI test rig, you can easily lead it towards this system. However, even the most limited GUI Toolkits can grow this system from their primitive components.

Tune your environment until you achieve each of these goals, in roughly this order.

- **One Button Testing**—the test rig takes care of all details
- **Just Another Library**—treat GUI controls as objects
- **Regulate the Event Queue**—learn how it works, then throttle it like a spigot
    - **Temporary Visual Inspections**—reveal a window in a tested state
    - **Temporary Interactive Tests**—drive it to spot-check responses
    - **Broadband Feedback**—abstract GUI review from manual interaction
- **Query Visual Appearances**—assert what the window looks like
- **Simulate User Input**—mock the user, and assert the window's new state
- **Fault Navigation**—tests, editors, and GUIs collaborate to illustrate faults
- **Flow**—the goal is rapid, heads-down coding without displaying the GUI.

Those ideas need more explanations, and technical examples.

One Button Testing

To test, keep your hands on the keyboard or mouse, and tap one function key or click one button. The button invokes a test rig to save all changed files, compile, run selected tests, and report the results back, as fast as possible, and with no further interaction.

Your IDE might not do all those things. The sub-chapter Least Favorite Editor, on page 386, illustrates the kind of "glue code" an IDE might need to support an application written in multiple languages.

Do not, for example, write some tests in one test rig, and some in another, and launch these rigs from command lines at change time. Research how to get all their aspects into one build script, and bind it to your editor.

That principle generates all the other principles. They are all reactions to common reasons we might hesitate before testing, or perform any other interaction after testing.

Just Another Library

When you learn a GUI Toolkit by learning to use its form painter, you may begin to think in terms of "GUI on the outside, source code on the inside". Event-oriented programming is not a good path towards event-*driven* programming.

To learn to take control of windows and forms "as objects", we set a very simple goal:

During a test run, permit the fewest possible obnoxious side effects.
Run as many test cases as possible without displaying windows.

That's just a goal; it's not a Principle. The attempt to suppress all window `Paint()` events generates useful test code. Most windows keep their state in an internal object model, so they can quickly react to inputs before they slowly paint their displays. When you harness that effect, and query back your control's internal state, your tests can run quick and quiet. And that, in turn, leads to less hesitation before hitting the One Test Button.

Some GUI Toolkits were designed to introduce programming, and they come with naïve conveniences that interfere with this principle. Some, for example, will throw an error if you attempt to perform trivial actions, like enabling a control or moving the focus while a window is hidden. Both those actions have memory representations; the GUI Toolkit needs no display hardware to accomplish them. Users would see the results the next time the window paints. Such bogus assertions might possibly help some junior programmers keep their windows visible.

Some GUI functions drive display hardware directly, and retain no impressions in memory. To test our code that invokes these graphic commands, we must first retain them. The research to treat the GUI Toolkit as a library will then support the research into its deviations.

Regulate the Event Queue

To take control of a window, under test, you must learn how its event queue dispatches messages. Some messages you must throttle—especially the paint messages.

And some messages you must let through, such as a test case's messages that simulate user input.

As test cases learn to control these aspects, they will grow tools to display windows under test. Write a function called `reveal()`, and call it from test cases, to obtain the next two Principles:

Temporary Visual Inspections

Turn on the Event Queue, during a test case, to visually inspect its window, populated by test data. The test run shall block until the window closes, then subsequent tests shall run.

Temporary Interactive Tests

While that GUI displays, drive it manually to research the GUI's behavior.

Those two temporary practices provide an alternative, test-side GUI. The `reveal()` fixture displays the window under test, populated with test data. That allows us to spot-check our test case, and adjust the test-side support for the other practices.

Then we comment out the `reveal()` call, and ensure the test code works without visual inspections.

The ability to temporarily `reveal()` leads to a convenient platform to record images of windows under test.

Broadband Feedback

GUIs can record animations of activities during tests. Acceptance Test Frameworks can command test cases to record a gallery of results.

The previous 6 Principles force the review of GUI appearance and behavior into the tests, away from the wizards, form painters, and finished applications. They allow us to take control of the situation, and prevent the need to excessively manually test.

Now we leverage those Principles to develop test fixtures (test-side reusable methods) that enable predictable changes.

Query Visual Appearance

GUI controls often provide the ability to query back their display values. All such queries are simulations. Some results aren't accurate! Tests must learn how to query that a GUI correctly obeyed the commands from its GUI Layer code.

- Tests on **Controls** may Get details which Production code Set.
- Tests on **Scripts** parse the script, looking for details.
- Tests on **`Paint()`** events use Mock Graphics objects to generate a Log String.

If a *Temporary Visual Inspection* reveals some difference of opinion between your assertions and your actual GUI, you need to research how to query more accurately. Invest this research into test fixtures that you can then re-use.

Simulate User Input

Test cases simulate events and check responses. This is harder than it sounds, because GUI Toolkits always treat signals from their programmer-side differently from their user-side.

The fix, generally, is to write a *Temporary Interactive Test* that investigates differences between the GUI Toolkit's available systems and a real user's input. Then new test fixtures can encapsulate these differences, making high-level tests easier to write.

Loose User Simulations

Call the same method as an event handler would have called.

Most of our Case Studies rely on this technique as it is low risk. Sometimes only simple logic binds a message, such as the targets of a WTL `BEGIN_MSG_MAP()` macro (see page 198). In these systems, the binding code is structure, not behavior. If you won't refactor `BEGIN_MSG_MAP()`'s innards, you have fewer reasons to strictly test that a message of type `IDOK` will indeed travel through the message queue and arrive at `OnCancel()`.

If an '`if`' statement, or a more complex expression, binds the event, then you must test this behavior.

Firm User Simulations

Write a test fixture that finds the handler bound to an event and calls it.

All GUI Toolkits come with the risk that they can Set a control's value, such as its bound event handler, without providing a matching Get to robustly query that handler. A GUI Toolkit written to support test-free projects does not understand why your code might try to Get the same value that your code just Set.

This situation may require extra research to learn just how to accomplish that tricky Get. The Tk library, for example, allows us to query a control's internal data structure (see page 120 for this technique). Our code must then navigate that structure to find the bound method reference within. And this code must still work when details that toolkit maintainers consider private change.

Coupling our tests to a GUI Toolkit's internal systems introduces the risk that our vendors may upgrade that system and break our tests. These and other risks may push our tests up this list towards *Soft User Simulations*, or down into the territory of capture/playback tools.

Strict User Simulations

Push a raw input event into the event queue.

If you research your GUI Toolkit's event queue, and learn its exact operation, you can often uncover its mechanism to forward raw inputs. Then you build fake event messages, push them into the event queue, and operate the queue so its other end dispatches events. Invest this research, as usual, into test fixtures that enable the kinds of fake input events that test cases for your application require.

This is how generic capture/playback tools test GUIs; by intercepting input events at the OS layer of the GUI Toolkit, and then by simulating these events in large batches. This "nuclear option" of GUI Testing should only be used after exploring the alternatives that permit test-first programming. You can't use capture/playback to test *first*.

No Case Study in this book needs such a high level of simulation. The following example code uses a rare and exotic language called “Java”, invented by James Gosling. The example reveals a GUI test rig called “Jemmy”, invented by Aleandre Iline. It exercises complete *Event Queue Regulation* for the Swing GUI Toolkit. The small penalty for these easy features is windows that flicker and animate their behaviors during tests.

Special thanks to Andrew de Torres for presenting this to the XP San Diego Users Group, and for allowing me to use it here.

The target of this test is a simple Swing form:

```
package src.com.ureach.detorres.jemmytest;
import java.awt.GridLayout;
import java.awt.event.*;
import javax.swing.*;

/*    This simple “Hello World” application demonstrates testing Swing applications
      with JUnit (by Kent Beck and Erich Gamma) and Jemmy. The application consists
      of a JFrame with a JTextField, JButton, and JLabel.  Initially the field is blank, the
      button is disabled, and the label is blank.  When you type “Enable button” into the
      field, the button enables.  When you push the button, a dialog displays. When you
      acknowledge the dialog, the label displays a status message, and the field and
      button are disabled.

 * @author  Andrew de Torres, detorres@ureach.com
 */

public class HelloWorld
{

    private JTextField textfield;

    private JButton button;

    private JLabel label;

    public HelloWorld()
    {
        textfield = new JTextField(20);
        button = new JButton("Push me");
        button.setEnabled(false);

        label = new JLabel();

        JPanel panel = new JPanel(new GridLayout(3, 1));
        panel.add(textfield);
        panel.add(button);
        panel.add(label);

        final JFrame frame = new JFrame("Hello, World!");
        frame.setContentPane(panel);
```

```
        textfield.addKeyListener(new KeyAdapter() {

            public void keyReleased(final KeyEvent evt) {
                if (textfield.getText().equals("Enable button")) {
                    button.setEnabled(true);
                } else {
                    button.setEnabled(false);
                }
            }
        });

        button.addActionListener(new ActionListener() {
            public void actionPerformed(final ActionEvent evt) {
                JOptionPane.showMessageDialog(frame,
                        "You're almost done.",
                        "Hi!", JOptionPane.INFORMATION_MESSAGE);
                label.setText("You're done.");
                textfield.setEnabled(false);
                button.setEnabled(false);
            }
        });

        frame.pack();
        frame.setLocationRelativeTo(null);
        frame.show();
    }

    public static void main(String[] args) {
        new HelloWorld();
    }
}

// eof "HelloWorld.java"
```

Jemmy solves a common problem simulating events. Suppose a button provides a `.push()` method, so production code can simulate user input. However, suppose that `.push()` method worked even if its button were disabled. When test cases rely on the `.push()` method, alone, its results might mislead. If a button was disabled and should be enabled, then if a test called `.push()` and recorded an enabled response, the test would not catch that bug.

This test uses wrappers, called Observers, which match simulated input behaviors to real input behaviors:

```
// HelloWorldTestUI.java

package test.com.ureach.detorres.jemmytest;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.WindowConstants;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import org.netbeans.jemmy.*;
import org.netbeans.jemmy.operators.*;
import src.com.ureach.detorres.jemmytest.HelloWorld;

/*  JUnit/Jemmy test for HelloWorld.  This class demonstrates the basics of testing a
    Swing application using Jemmy inside of JUnit.
```

```
 * @author Andrew de Torres, detorres@ureach.com
 *
 */

public class HelloWorldTestUI
    extends TestCase
{
    public HelloWorldTestUI(String testName) {
        super(testName);
    }

    class Flag {  boolean flag;  }
    final Flag flag = new Flag();

    public void reveal(final JFrameOperator frame)
            throws InterruptedException
    {
        frame.setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);

        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                flag.flag = true;
                frame.removeWindowListener(this);
            }
        });

        while (!flag.flag)
            Thread.sleep(100);

    } //  by Timothy Wall

    /**
     * Our single test method which tests everything.
     */
    public void testIt() throws InterruptedException
    {
        // Turn off Jemmy output.
        JemmyProperties.setCurrentOutput(TestOut.getNullOutput());

/*  Because Jemmy operates a real event queue, it must await streams of messages to
    communicate between the platform and the GUI Toolkit. Some controls won't finish
    painting until after an indefinite number of messages, so Jemmy relies on small timeouts.
    For each kind of control, test cases must declare how long they will wait before grabbing
    that control. A complete test rig would configure these defaults in a fixture such as
    setUp().
*/

        // Shorten timeouts so things fail quicker.
        Timeouts.setDefault("FrameWaiter.WaitFrameTimeout", 5000);
        Timeouts.setDefault("DialogWaiter.WaitDialogTimeout", 5000);
        Timeouts.setDefault("ComponentOperator.WaitComponentTimeout", 5000);

        // Start the application under test.
        HelloWorld.main(new String[0]);
```

```
/*    Jemmy wraps each kind of target control in an Operator object. This test fixture mediates
      between test cases and simulated user inputs.
*/

        final JFrameOperator frame = new JFrameOperator(
                                                "Hello, World!");

        // Find 1st (0th) JTextField in frame.
        JTextFieldOperator textfield = new JTextFieldOperator(frame, 0);
        assertTrue("textfield is enabled", textfield.isEnabled());
        assertTrue("textfield is editable", textfield.isEditable());
    //    reveal(frame);

        // Find "Push me" button.
        JButtonOperator button = new JButtonOperator(frame, "Push me");
        assertTrue("button is disabled", !button.isEnabled());
        JLabelOperator label = new JLabelOperator(frame, 0);
        assertEquals("label is blank", "", label.getText());

/*    If this button.push() were not commented out, it would do nothing. Jemmy Operators
      wrap each kind of target control in an Operator object. This test fixture mediates between
      test cases and simulated user inputs.
*/

      //  button.push();

        // Simulate typing text.
        textfield.typeText("Enable button");
        assertTrue("button is enabled", button.isEnabled());

        // Simulate pushing button.
        button.push();

        // Note partial match - dialog title is "Hi!".
        JDialogOperator dialog = new JDialogOperator("Hi");
        new JButtonOperator(dialog, "OK").push();
        assertEquals("label changed", "You're done.", label.getText());
        assertTrue("textfield is disabled", !textfield.isEnabled());
        assertTrue("button is disabled", !button.isEnabled());

        // Throw in a dispose here incase we run with
        // the JUnit GUI (see main method).
        frame.dispose();
    }

    public static Test suite()
    {
        TestSuite suite = new TestSuite(HelloWorldTestUI.class);
        return suite;
    }

/*    The main() function runs the test using the JUnit test runner. If the single argument
      “swing” is specified, the GUI test runner is used. Otherwise, the command line test
      runner is used.

   * @param _args command line args: ["swing"]
   */
```

```
    public static void main(String[] _args)
    {
        String[] testCaseName =
                    { HelloWorldTestUI.class.getName() };

        //      _args[0] = "swing";

        if (_args.length == 1 && _args[0].equals("swing")) {
            junit.swingui.TestRunner.main(testCaseName);
        } else {
            junit.textui.TestRunner.main(testCaseName);
        }
    }
}

// eof "HelloWorldTestUI.java"
```

An off-the-shelf GUI test rig should provide a balanced set of generic fixtures that cover common GUI aspects. Even so, our case `testIt()` is very long. As we add more tests to this project, Extract Method Refactor will merge code out of long cases, and grow new, reusable, application-specific fixtures. New test cases, for this specific application, will become easier to write.

Ultimately, user simulations can replace a user with a Mock Object that generates a complete sequence of input events. Page 305 shows how to record a trace of user inputs (from a *Temporary Interactive Test*, naturally), and assemble them to form a "Motion Capture".

If a *Temporary Interactive Test* reveals some difference of opinion between your simulations and your actual GUI, you need to research how to simulate more accurately. Invest this research into test fixtures that you can then re-use.

All that test infrastructure must interact with your production GUI, your test code, and your editor, in ways that rapidly identify the sources of errors.

Fault Navigation

After a test run, the programmer can tell if a compile fails, or a test fails, or the code faults, without pressing any other button besides the One Testing Button. After a failure, the editor optionally automatically navigates to the failing line.

This Principle requires collaboration between your editor, environment, GUI, and test rig to bubble expressive error messages up properly, without interference. Your editor and test rig should collude to treat assertion failures the same as syntax errors, so you can rapidly and optionally navigate to them.

Flow

During tests, the editor remains available, activated, and unblocked. You just keep typing, and you don't change your programming behaviors when you upgrade GUI code. It's all just code.

The rhythm of testing drives development. Predicting the result of each test run reinforces your mental model of the code's state. Integrating, passing acceptance tests, and releasing upgrades reinforces a team's mental model of a project's state.

Any encumbrance in our test rig will cause hesitation before hitting the test button. That, in turn, will impair Flow. As you develop test support for the other TFUI

Principles, make certain their practices do not create obnoxious side-effects that lead to hesitation. Use Flow to enable more Flow!

Conclusion

GUI developers should focus on these problems early, and find optimal solutions for the intersection of their GUI Toolkit and their target application.

Many of these ideas may seem obvious. However, for every GUI project that easily solves one, another misses it completely. The idea of an editor that remains unblocked during a test run might seem impossible to programmers familiar with, say, Visual Basic 6.

Many of these ideas may seem *too* obvious. Your GUI Toolkit might spontaneously solve some problems for you. For example, the TkCanvas makes *Query Visual Appearance* very easy, by treating graphic primitives as objects with properties. You should correlate the easy principles with the others. The TkCanvas can occasionally Set a value different than it can Get. *Temporary Visual Inspections* will rapidly and easily catch these issues.

Put together, the strategy seems to resemble a paradox: To rely on tests, and avoid manually reviewing GUIs, you must write a `reveal()` method that manually, temporarily reveals a tested GUI, for review. This is not a paradox; it is taking control of the situation. Wizards, form painters, and debuggers no longer control us. We control when and how to manually review our GUIs. When test cases temporarily present GUIs in their exact tested state, differences between those GUIs and the tests' opinions become easy to fix.

The strategy works for any GUI, so the next chapter converts this Principles into specific practices for the most common GUI architectures.

Chapter 3: *GUI Architecture*

The previous chapter listed some ambitious Principles that work for "every" kind of GUI Toolkit. This chapter relates the Principles to specific GUI practices.

Most books on GUIs specialize on a single toolkit. This one shows how their common architectures fit our testing patterns. Every GUI is different but every GUI is the same. As mentioned before, each GUI Toolkit uses a mix of three specific systems:

- **Controls**—objects that encapsulate data animations
- **Scripts**—light programming languages that author controls
- **`Paint()` Events**—the low-level graphics commands that draw controls

Developers build windows and forms by arranging controls on them. They create the controls as objects within a language, or they write scripts to create controls, from a secondary language.

When users click on controls and enter inputs, GUI Toolkits package information regarding each input together into an event message, and route these messages to event handlers. GUI Toolkits consume most events internally, to supply controls' default behavior and animations. Programmers connect some events to event handling functions, to route new data into an application and respond to the user.

The `Paint()` event calls its handlers when the user raises, uncovers, or changes a window. It calls low-level graphic commands to paint a control in its current state.

Developers create custom controls by packaging prefabricated controls together, and linking their event handlers. The most common example is the lowly "combo box", combining an edit field and a list box.

Developers customize some controls' appearances by providing a `Paint()` event handler, and changing its graphic commands.

All GUI Toolkits mix and match some combination of these three systems. Sometimes they route input and output in different systems. Many desktop GUIs use scripts to generate controls, and their input events directly call methods bound in code. HTML, by contrast, uses scripts to generate remote controls, and their events return as HTTP messages containing script-like blocks of text.

Your application's architecture must adapt to some mix of these methods, while isolating as much logic as possible into a unified Representation Layer.

An Ideal Layering

To illustrate an optimal architecture, we study the lifespan of a single variable through a hypothetical system. We route it from a database to the screen, then receive inputs that change its value in the database.

Output

Our variable—call it "z"—travels from a database to three screen locations. The user sees it as a value in an edit field; as a value presented inside a scripted display, such as an HTML page; and as a color in a custom control, such as a color map. Figure 1 illustrates this variable's journey:

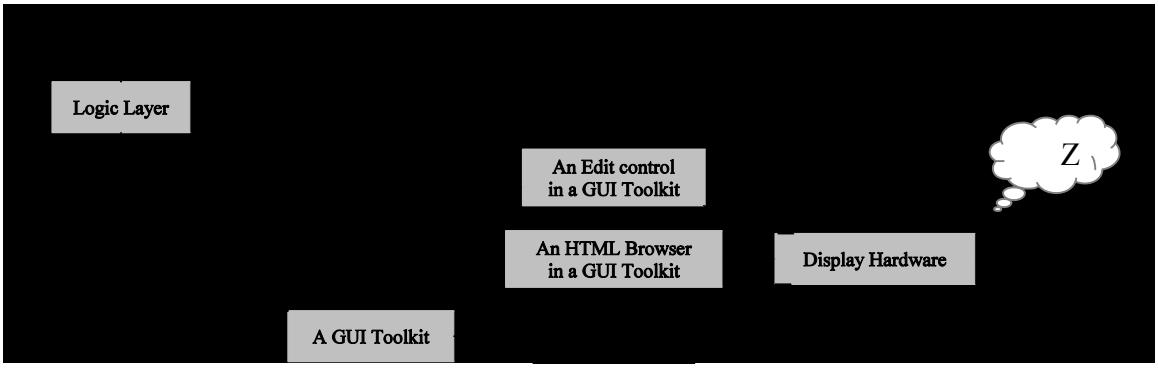


Figure 1: An ideal output layering

In that diagram, the modules with thick frames might upgrade while we route that variable to the user. Changing the contents of the darker Logic Layer modules, GUI Toolkits, or display hardware should be separate activities. This book treats them as opaque.

The variable begins life somewhere in the regions that other texts might call the "Business Objects Layer", or the "Persistence Layer". This text lumps all such locations under the umbrella term "Logic Layer". All we care is that something generates `z`, and maintains its logical coherence.

Our ideal system processes variables through a Representation Layer before display. A "Layer" needn't be heavy-weight. Our `z` might translate into user terms using a simple function. For example, we might convert Z from centimeters (an ISO standard) into locale-specific inches. A thin GUI Layer delegates many user interface responsibilities. To preserve flexibility, only our Representation Layer may access our Logic Layer, and only our GUI Layer may access a GUI Toolkit.

Our diagram's GUI Layer pushes our variable `z` into three different screen locations. To put `z` into an edit field, that Layer commands the GUI Toolkit to create a window and draw a control on it. Then the GUI Layer **Sets** the control's "value" property to `z`. All GUI Toolkits have many control types—edit fields, list boxes, pushbuttons, etc. Every Toolkit provides a common system to manage each control's long lists of display aspects. Controls follow the Variable State Pattern, using systems variously called Properties, Configurations, or Attributes. Methods with persistent side effects, such as `Move()`, also affect those display elements. This book, as an abbreviation for all of these diverse manipulations, often refers to methods that transfer state into a control as Set.

The edit field now remembers `z`'s value, and when the GUI Toolkit tells the edit field to paint its contents on the screen, it will format that value, in the correct font, location, color, etc. These primitive commands go into a Graphics Layer (not shown), which is little more than a driver for the Display Hardware.

To Set the variable `z` into an HTML page, the GUI Layer inserts it into a string containing properly formatted HTML tags—`<html><body><table>`, etc. Then the Layer submits the entire string to some control or browser that can display HTML. This widget remembers the whole

string, and when *its* GUI Toolkit tells it to paint its contents on the screen, it formats that value, in the correct font, location, color, etc.

Figure 1 presents "HyperText Markup Language" as only the most common example of dozens of scripting and encoding systems—RC, RESX, RTF, SVG, UTF, XUL, etc.—that intermediate between a programmable GUI Layer and the user's experience.

In Figure 1, `Z` also arrives on the screen by a third route. When GUI programmers need to display a variable in a format that no available control provides, they assemble this format out of primitive painting commands, generating a custom control.

When GUI surfaces peacefully coexist among overlapping windows, other windows determine when our window needs to repaint. For arbitrarily complex displays, programmers write a "`Paint()`" method and register it with the GUI Toolkit. When the host window is ready, the Toolkit calls each of its clients' `Paint()` methods. Some systems call this event `OnPaint()`, and some Toolkits pass the integer constant `WM_PAINT` into a generic handler, which then dispatches to such a method. All `Paint()` events pass into their handlers a "graphics device context", such as a GDI `hDC` handle, a `PaintEventArgs` object, or a `Graphics` object. These mediate commands through the graphics layer into the display hardware.

To render `Z`'s value, statements inside the `Paint()` event handler call primitive methods with this handle or on this object. We might convert `Z` into a color value, `clr`, and pass this into `SetColor(clr)`. Then `RectFill(x1, y1, x2, y2)` might draw that color into some rectangular area. Different Toolkits, naturally, use different method selector names, but they all behave like a paint program, with a toolbar of different brush types and a palette of colors. Display hardware stores the result, typically as a rectangle of colored pixels called a "raster". Sequences of painting operations accumulate to generate an arbitrarily complex picture.

When possibilities are endless, so are responsibilities.

Most `Paint()` event system use non-retained graphics. Each time the GUI Toolkit needs the control to paint again (maybe after the user maximized its host window), the Toolkit calls its `Paint()` method. After the `Paint()` method returns, the GUI Toolkit efficiently remembers nothing. The graphics handle or object delegates directly into the display hardware. The only side effect is different colors stored in some of the pixels of the display hardware's volatile memory. When another window moves over them, its colors overwrite ours.

When our window returns, or when the Toolkit registers any similar change to the GUI's state, the Toolkit calls `Paint()` again. The Toolkit did not, for example, remember any calls to the primitive graphics object and replay them. (Some windowing systems *do* remember a copy of the raw pixels, and temporarily refresh by copying these back into the display. That optimization prevents a screen flicker before our slower `Paint()` event provides the real refresh. The appearance might then change. Our tests should never interfere with such internal details.)

If we look inside a control or a script viewer, they obey similar `Paint()` events. When the user obscures and then reveals them, their `Paint()` methods **Get** `Z`'s value, among other values, and convert them into a series of primitive graphics commands.

Both the edit field and HTML browser must store intermediate display values, to supply their `Paint()` events.

Many controls provide an "owner draw" system for some component of their display. They provide the option to send a message to their owner window, requesting its methods to assist painting. When programmers write these owner windows, they can customize some aspects of a control's appearance. These requests are `Paint()` events too. Programmers implement their event handlers, and concoct graphics commands based on data stored inside the control, and on data from the program's state.

All GUI Toolkits provide some combination of those three channels. All provide controls, and most provide scripting languages, that channel Set configurations into their `Paint()` event handlers. And most let programmers write their own `Paint()` event handlers, to build custom controls from graphics primitives.

When you develop a GUI, prefer a control unless you must script. If you need custom controls, prefer **Aggregate Controls** that package primitive controls together. If no control animates data the way you need, use a full-featured canvas that treats graphic primitives as objects, not painting commands. Override a `Paint()` event only as a last resort, or to adjust the display of an existing control.

Prefer controls to scripts or raw painting primarily because they offer high level interpretations for user input.

Input

The hypothetical journey of our variable, `z`, is far from over. When the user wants `z` to change, they will click on or type into one of the three kinds of controls. They manipulate their keyboard or mouse, while thinking about `z`. (GUI designers must remember that experienced users don't think about the keyboard or mouse independently.)

The keyboard and mouse create raw electrical signals, and hardware drivers translate these into input events relevant to programs. For example, a moving mouse only transmits a series of primitive electric ticks. A driver counts the ticks, and measures the times between them, to calculate the mouse pointer sprite's new screen position.

The cooked event now enters our GUI Toolkit, and joins an event queue. The Toolkit processes the queue, performing more validation and sanitation, before passing the event into an event handler in our GUI Layer.

All GUI systems maintain the concept of an "active" window and a "focused" control, meaning users move a single focus point around their screen—typically with <Tab>, <Alt+Tab>, <Arrow> keys, or mouse clicks. Controls reflect the focus point's location by drawing a "selection emphasis". This can be a dotted box around their labels, or a text caret ("cursor") inside a control's edit field. When users type on a keyboard, the keystrokes "go into" the currently focused control in the currently active window. This means the GUI Toolkit must route input events based on the desktop's state. Out of any number of different applications' windows, only one control has the selection emphasis or text input caret.

Even then, such inputs might not go all the way to the program as high-level commands to change the `z` variable. The user might have only thought about changing `z`, then decided to do something else. If, for example, the mouse pointer drifts over `z`'s control but performs no confirmed action, the Toolkit discards many of these events.

When the user decides to change `z`, they enter some input events, and the control changes its state to display the new potential value. Then users supply a "turnaround event" to commit the changes, or an "escape event" to return the control to its previous state. At commit time, the control's type indicates what happens next:

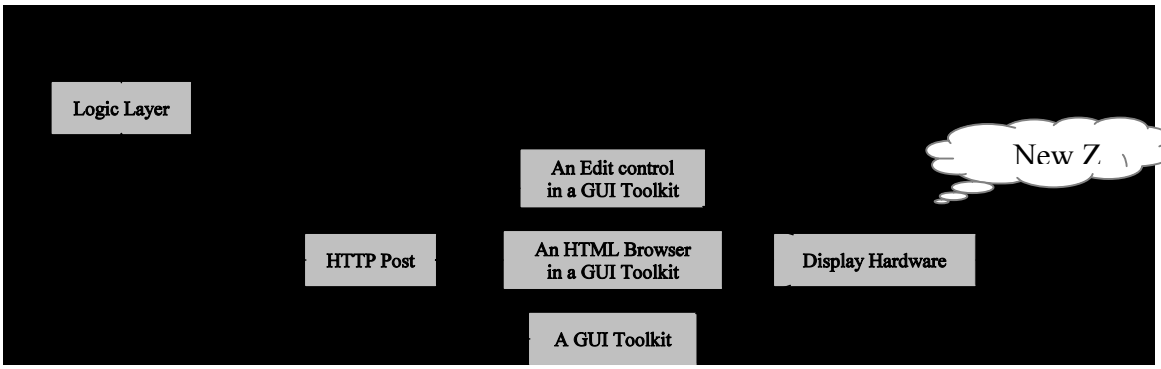


Figure 2: An ideal output layering

All GUI programs are **Event Driven**; they await input events, on one or more channels, and convert them into changes of internal state. Some events are high-level and definitive, and some are low-level and ambiguous. Programs must decide which inputs to ignore and which to obey.

(The best architectures for the Representation and Logic Layers are also event driven. Events come from either their tests or their associated layers. Such event queues should not be asynchronous. Each event updates an internal data model. The shape and state of this model then generates a module's responses.)

An "event handler" is a function or method that you write, but someone else calls. GUIs work by operating one message pump, with names like `mainloop()`, `GetMessage()`, `Application.Run()`, etc. This pump collates events for all windows and controls. Control flow enters `mainloop()`, and does not return until the program exits. (MS Windows' `GetMessage()` works a little differently; one could write Tk's `mainloop()` using `GetMessage()` inside it.) Each control's input events route through that method, and it dispatches them to each control. Controls may pass an input event to a bound event handler. This Observer Pattern enables your code to observe the user's activities.

The Event Queue

All GUIs support an event queue, whatever its architecture, that collates and sequences output and input.

The `Paint()` event handler responds to the user's act of revealing a window or control by rendering its pixels. Most windows and controls provide built-in `Paint()` event handlers, and you should not need to touch them.

Your GUI Toolkit provides default callbacks for most simple inputs. Some dialogs, for example, are windows with default actions for the keystrokes <Enter> & <Escape>, and for their equivalent command buttons.

When a user changes `z`, edit fields can often send very high-level events. When they are aware of simple types—strings, integers, etc.—they can return `z`'s new value in its native format. If not, the GUI Layer and Representation Layer may collude to change the format back. Either way, such controls provide a very high-level awareness of `z` as a variable. Some GUI Toolkits can refer to variables in the GUI Layer's memory, and change them directly.

GUI Toolkits often augment their form painters with a "Class Wizard", to manage lists of links between controls and their event handlers. In our parables, form painters and wizards will often play the role of the "Big Bad Wolf". These tools are not the problem. Enabling

programmers to neglect tests is the problem. Beefing tests up with application-specific features that exceed generic form painters and wizards' ability to propel development is the solution.

Some scripting systems generate controls that a GUI Layer may address as objects. Our `z` variable might travel to the screen inside a script, but return as the value of an edit field.

Many scripting systems, most notably HTML, also operate remotely, as thin clients. HTML transmits into Web browsers on the "HyperText Transport Protocol". Browsers handle all low-level user input events, until the user Submits a completed form. The form's results come back through a server into our application. They arrive in HTTP Post messages that are the moral equivalent of text e-mails. This slow, low-bandwidth connection strips out all extraneous events, and only sends batches of variables in their new committed states.

HTTP is also an event queue; tuned for low bandwidth and high latency.

The third system to change `z` is far more complicated. If you use a `Paint()` event to concoct a new kind of control, from scratch, then you must process too many input events, instead of too few. Only your own code can decipher the stream of primitive events from the GUI Toolkit. Your handlers reflect some events into selection emphases, and interpret which sequences represent the user's attempt to change `z`. Custom controls must tune for high bandwidth and low latency.

The Representation Layer might respond to `z`'s changes by pushing its new value into the other two screen locations, again following the Observer Pattern.

Ultimately, all three of these systems commit a new value for `z` into the Logic Layer. All three require GUI Layer code to Get a new `z` value, by retrieving some Property, Configuration, Attribute, or low-level input event. And all three input systems require tests that Get incidental display aspects from those controls, to ensure no control leads users astray.

Best of Both Worlds

Agile development depends on test-first programming to make changes safe, and merciless refactoring to grow architectures as needed.

GUI development, by nature, resists testing. Most GUI bugs are easy to spot and easy to fix. Most aspects of a GUI need no tests, only authoring. Contrarily, upgrading GUIs can quietly change their appearance, leading to bugs. How can we bridge these gaps between the creative world of GUIs and the precise world of constrained development? How can tests assimilate GUIs?

Internal logical modules are not so easy to spot-check, so their developers aggressively pursue the goal: "If it doesn't have a test, it doesn't exist." We can easily see if GUI features exist, so they can follow a softer rule, "Test everything that must work." By "work", we mean everything that must not fail late, quietly, cryptically, or irreversibly.

In a GUI, if you author some detail, such as a blue border, a failure is easy to spot and easy to fix. However, if that detail comes from internal logic, a failure might be hard to spot. Suppose that blue border comes from an internal temperature setting, and you spot too late the border is orange when the temperature is cool. You might spot this problem long after upgrading the internal logic, so you can't just Undo to back out the change that caused the bug. Tests can't fail reversibly unless they fail early, loudly, and expressively.

All GUI testing falls into patterns depending on the target architecture.

To Test Controls…

To test **controls**, exploit their object model in memory. GUI Toolkits invariably represent these controls as a navigable Composite Pattern of objects in memory, each with a Get and Set system for each control's list of properties.

All code samples in this book start with that technique. The chapters Chapter 5: *SVG Canvas*, on page 68, and Chapter 6: Family Tree at 116 use a simple but powerful control, TkCanvas, to push this technique much farther than may GUI Toolkits permit, into the realm of interactive and editable canvases.

The chapter Chapter 11: *Fractal Life Engine*, on page 307, uses the Qt library, which makes that technique so easy the chapter does not bother to illustrate it.

Your GUI problem, and your toolkit, might require more rough techniques.

To Test Scripts…

A script is a light programming language that can embed inside an application. Most GUI scripts generate controls. They become easy to test, using the previous techniques, if tests can reach out to the application running the script and get to the controls' representations as objects.

If tests cannot, or should not, reach out to the effects of scripts, tests can parse the script in its native format, and can query out details. If you write a web site using XHTML, you can spot-check its contents using XPath. Chapter 12: *The Web*, on page 365, naturally relies on this technique for most tests.

If you do not have a parser for your language, and if your scripts are high-risk, use TFP to write a parser. Some teams might consider writing a parser just to write a test wasteful. Such parsers needn't apply a script's entire grammar; they need only look for the target details. And whether you use a high-level parser system like Lex and Yacc, or a generic parser like Regular Expressions, or a brute-force parser with `char *` and `strncmp()`, TFP will support this effort and help your parser grow with your tests needs.

To Test Paint() Events…

GUI features that don't store state cannot test that state. One cannot (economically) write a test that asks what a window "looks like".

These are the hardest techniques, so try to get by with the former ones. Only attempt these techniques to prove you can, or to constrain advanced situations. If you build a new editor from scratch, start here.

All tests on `Paint()` events use Mock Graphics, described below. However, many GUI Toolkits make some `Paint()` events deceptively easy. Test-first a Paint() event if you invent a new control from scratch, or need to tune a control's appearance.

Our chapters which do not test `Paint()` events illustrate many alternatives to creating custom controls from scratch. You can assemble ready-made controls, the way a "Combo Box" combines an edit field and a list box. To paint arbitrary shapes, use a canvas control that abstracts its `Paint()` event.

As these alternatives run out of steam, your team must decide whether to author `Paint()` events, or mock them. Most simple tweaks to `Paint()` events, such as the technique to introduce tab characters into list boxes, only require applying sample code from your GUI Toolkit's documentation.

The sub-chapter GDI MetaFiles, on page 234, makes a test-first example of that situation, turning an elaborate `Paint()` event into a simple Log String Test.

When you invent a new, high-performance control from scratch, you need to invest in the effort to learn to test-first its `Paint()` event. Chapter 11: *Fractal Life Engine*, on page 307,

presents a technique to mock an entire dynamic link library, at its interface. This powerful technique will mock many more kinds of `Paint()` events than rotating fractals rendered in OpenGL.

Mock Graphics

A Mock Object is a test fixture that production code thinks is a real application object. Imagine hunters placing decoys of ducks into a pond. Real ducks obey their instinct to flock with things that look like ducks. Decoy ducks have no meat inside. Similarly, a test may construct a decoy that looks like a real object, and place it into the pond of production objects, to see how they interact with it. Mock Objects have little or no logic inside.

Some Mock Objects automatically generate any methods their clients request. In our hunting parable, the ducks fly away, and the decoy sprouts wings to follow them. Some Mock Objects take a reference to a real object, to mock all its members. In our parable, a biomimetic decoy resembles any required gullible social creature.

An emulator, by contrast, has enough logic inside to simulate the meat of the represented object or system. Think of a full-featured, robotic duck decoy.

If production code must call a sequence of methods on the Mock Object, it can record each one into a Log String, and the test case can then assert the sequence was correct.

A `Paint()` event handler, to paint a custom control or customize an existing one, typically dispatches a sequence of graphics driver commands. To constrain the event handler, we must mock the graphics driver.

Here's an example of this technique, in fictitious Java:

```
public TestGraphs extends TestCase
{
    public void testDrawLine()
    {
        Graph graph = new Graph();
        Graphics mock = new Mock(Graphics.class);
        graph.Paint(mock);
        assertEquals( "drawPolyLine(200, 100, 100, 200);",
                      mock.getLogString() );
    }
}
```

It implies the `Paint()` event drew a diagonal line. However, if that event called more methods, the Log String would grow long and fragile. This technique trades an impossible situation—mechanical review of some inscrutable `Paint()` method's rasterized output—for a very hard situation, a "Parsed Fuzzy Match".

Use Mock Graphics as sparingly as you invent custom controls. Prefer full-featured canvases, such as TkCanvas, that treat graphical primitives as objects with complete Get and Set systems. If you must `Paint()`, select an existing control that permits you to override only the graphical aspect you need to improve.

Some graphics drivers provide the option to record all their commands to a file instead of display them; these systems can grow into Mock Graphics fixtures. On page 234, we customize the appearance of a list box, constraining its appearance using MetaFiles to retain GDI graphics commands.

Some systems come with no option to record their commands. On page 324, we write a system to mock an entire dynamic link library, at its raw interface, to generate a Log String and constrain 3D graphics.

Log String Tests can make test-first a little fuzzy. If a test breaks, but visual inspections confirm the appearance is correct, one can pass the test by copying the output Log String into the test's reference Log String. As a project grows, new test cases will cheat less often. Test fixtures will learn to constrain features instead of strings, and new cases will reuse these fixtures. Cheating happens most in the early phases of a project, while TFP is still hard.

As you use more Mock Graphics, you will write more test fixtures that support Fuzzy Matches on their Log Strings. Ideally, sloppy refactors that change the order of graphics commands, without changing appearances, would not fail these tests. Test fixtures could interpret the commands and ensure the result still looks the same. But another fix for this problem is to not refactor sloppily. Remember to keep all your ducks in a row.

Hyperactive Tests

Because we are developing, not assuring quality, tests that fail too often are less risk than those that fail not often enough. Running tests after every few edits will force the review of low-quality tests.

Fuzzy Logic

This chapter recommends test cases that might run too far from the tested logic. Because low-quality tests will not last long, we need to make tests cheap and easy to write. Premature accuracy is a potential waste of time, just as premature optimization is. Rapidly author tests, and inspect frequent failures for opportunities to upgrade their assertions, just enough, to match requirements. If cheap tests pass, you win, and if they fail, you right-sized the effort to tune them.

But don't always blame tests. The strict definition of refactoring is changing design while keeping behavior the same. GUI code must not suffer sloppy "refactors" that change appearance. Legitimate changes must not break tests that constrain appearances.

All GUI Toolkits make changing features absurdly easy, so esthetics and logic upgrade independently. To change the order of two buttons on a form, one need only Set their locations to different coordinates. Agile projects don't need this kind of flexibility. Such projects must release working versions to users early and often, and these people will not appreciate the need to re-learn any GUI interfaces, just to get their jobs done. Most new releases should run faster with more features under the same interface.

A naïve test on a GUI might accidentally prevent a harmless esthetic change. Worse, a test on a script or `Paint()` event might fail hyperactively, with no esthetic change at all.

Write such tests with naïve abandon! Because everyone must run all tests relentlessly, frequent test failures can be adjusted and reduced over time.

GUI Toolkits make esthetic details easy to program, and easy to change. Any window will have far too many details (colors, widths, gutters, labels, etc.) for test-first to constrain them all. Agile development leverages frequent reviews as well as tests, so display bugs shouldn't live long.

If a visual feature could catch a bug that would mislead users, develop the feature using test-first. If a GUI Layer generates complex appearances, develop them using test-first. And if a GUI Toolkit makes some feature difficult to test, greet that challenge with research toward test-first.

When GUIs change too easily, a sloppy refactor or a new feature might change existing code in ways that affect appearances. Test-first must lead to tests that fail when their support code changes in an invalid way, even if the resulting appearance would remain the same. Tests

that constrain visual appearances can often be hyperactive if making them exact would be much more expensive.

When a programming shop uses test-last, and infrequently runs their suites, their hyperactive tests that constrain GUI details will break frequently. This leads to GUI testing's reputation for fragility. The more often hyperactive tests run, the more incentive programmers feel to polish these tests and improve their accuracy. (And the less incentive they feel to make sloppy refactors.)

GUI Toolkits will fight back in devious ways. Objects have state—member data variables—and methods that Get and Set that state. The most common test case Assembles a window with a control, then Activates production code to Set a variable into that control. The case then Gets that control's variable, and Asserts that it's within a valid range.

A Get method might be inaccurate, or missing. That's where the fun begins.

When production code Sets a value into a control, the GUI Toolkit commits to reflecting that value on the screen properly. Retrieving that value accurately, in a Get statement, is secondary. A control may merge several Set values together and display them, but provide no way to Get their conjoined value.

GUI Toolkit developers might not recognize an inaccurate Get method, or a missing one, as a bug in their systems. For example, consider MS Windows' primitive method `TextOut()`. It paints a text onto a display device, and does not retain any report of what it did. The display context has a font, and the text must contain characters matching glyphs in the font. If a glyph is missing, then `TextOut()` will paint a little □ spot; the "Missing Character Glyph". The interaction of the Set text value and the Set font value provides these.

However, `TextOut()` has no return value to indicate how many of those spots it printed. The function's innards know this information, but they don't retain it or return it.

When a system misses a Get method, extra researching into related systems might discover a way to Get an indicator for that method. (Open source GUI Toolkits also permit a less subtle attack…). Page 262 illustrates borrowing a method from another library to automate Missing Character Glyph detection.

The hardest situations we will cover are custom `Paint()` event handlers. They use Mock Graphics (described on page 33) to retain the graphics that normal display devices would throw away. Page 234 describes a system to provide a Log String by turning on a display device's graphics retention system.

Ultimately, the cost of some visual appearance queries will exceed their benefit. Agile processes leverage frequent reviews of appearances and usabilities. No matter how many test fixtures we put online, complex GUIs need frequent manual inspections.

Acolyte Checks Output

Projects that lack tests often use an AntiPattern called "Guru Checks Output". This means a complex module creates output, and this passes to an expensive guru who slowly reads it and verifies it. Computers exist to automate such activities! If a programming team provides an Acceptance Test Framework, then this guru can concoct test resources. Tests check the output, the cost of each test run goes down, the frequency of testing goes up, and everyone's time frees up for more important activities.

But to check every authored GUI detail, we make use of a resource more available (and faster) than a guru. As we code, we often look at our windows ourselves. If any controls overlap, or any splitter bars expand to the wrong proportions, or any text inexplicably turns upside down, we will quickly notice.

Such a bug leads to a judgment call. Should we just fix it? Or should we add a new test case to catch this kind of bug? Recall that non-GUI situations *always* require tests to catch bugs whose effects are remote and obscure.

The answer is an engineering trade-off. If a simple typing error obviously caused the bug (such as a **Bogus** command to the geometry manager forcing it to overlap controls), we just fix it. If we encounter any trouble reproducing the sequence of events leading to the bug, or trouble isolating the root cause, new tests will help.

A test rig to support catching bugs must already be in place.

Even when the problem is trivial, if we cannot easily see a way to test the given problem, our process could be at fault. Every kind of feature in our system should have a test rig available with sample tests, and we must research how to query our visual aspects efficiently. The solution, when we detect such a process fault, is to (you guessed it!) write more tests.

Within a healthy process, people occupying several roles review GUI appearances:

- engineers using Temporary Visual Inspections and Interactive Tests
- engineers manually testing their **Production Builds**
- engineers using debuggers, wizards, and form painters
- Customer Team roles (quality control, usability experts, etc.)
- users.

The elements of a GUI Toolkit—the controls, online help infrastructure, support for internationalization, etc.—already decouple from your own logic. Our tests pin down the logic, constrain the art, and enable refactoring.

However, tests should gracefully permit code changes that intentionally change visual appearance. Projects should be continuously ready to release.

Programming vs. Authoring

Though new test cases should always launch major features, not every code change requires a test. When you position controls on a window, abstract resources into locale-specific files, or raise balloon help and context-sensitive online help from each control, you are authoring. Failures in such features can hardly go unnoticed, or affect **Control Flow**, or cause bugs elsewhere, or lead to long bug hunts, or resist fixes. Test such display-only features occasionally, to prove you can, or to enforce elaborate requirements. Don't straightjacket art.

We test things liable to break. Changing code tends to break it. Change happens when code receives new features or better designs. Better designs typically de-couple. De-coupled modules can easily change at different rates. Your GUI Toolkit's source changes less often than your application. That toolkit lets you move a button without changing its event handler. GUI Toolkits decouple authoring from programming, to make the former easy and automated. Authoring represents rapidly changing a window's configuration, to put controls here or there, without changing logic.

To author, select a test on the window you need to change, and activate a Temporary Interactive Test. Then, instead of editing and testing in short cycles, edit and view the window in short cycles.

You are the test, so only risk failures that can't easily or silently harm the internal data. A mistake such as an edit field reflecting password stars *** instead of clear text may create a mere Display Bug—unless that field were rare, or some internal variable triggers the password feature. When you author you typically Set new constants into controls' lists of properties, without programmatic intervention.

Production code that adjusts default GUI behaviors requires test safety. Suppose your application localizes with pluggable resource files, one per culture. You substitute a common glossary of strings into all the source resource scripts, and generate distinct resource modules from each set of substitutions. This feature will only generate a few tests. They can't cover every string in every locale. Then suppose a linguist requests that the same string in a different location translate differently. This requires an "`if`" statement in the code. And that, in turn, requires a test that the correct alternate string appeared.

Small changes and Temporary Visual Inspections help resist errors. For example, on page 266, while authoring a new feature, we use a form painter to author a simple progress bar onto form. Then when we run the program it crashes. The form painter did not save us. Frequent testing tells us which control caused the crash – the most recent one – leading to a rapid fix.

Fuzzy Matches

When esthetic graphic elements need tests, these might require "wiggle room". Some tests for "red" should pass "fuschia" or "magenta". As you develop, if a hyperactive assertion fails, pick a fix from this list:

- Undo your most recent code change
- Remove the assertion
- Make the assertion match the code
- Make the code match the assertion
- Make the assertion fuzzy.

Don't make the assertion match the code, or vice versa, more than thrice. The pattern of hyperactivity will reveal how to provide the right kind of fuzziness. Page 113 shows code adjusting its behavior within hyperactive constraints, converting their hyperactivity into more accuracy.

The fuzziest test for Red would ensure the Red component of an RGB value exceeds the Green and Blue components. This would pass Brown.

Institute fuzziness when you predict you'll need it, when hyperactive assertions reveal you need it, or when a Design Smell in the tests, such as duplication, reveals the need for it:

```
assert( color == red or
        color == fuschia or
        color == magenta )  #  that's a smell
```

A test that the color occupies the purple, red, orange or yellow side of the color wheel, not green or blue, would give you the warm fuzzies.

Regular Expression Matches

Given a string, you might only care it contains the words "Sandy" and "Little Orphan Annie". The text between could localize, or reflect user input, or even just betray programmer whims.

```
assert_match(/\bSandy\b.*\bLittle\w+Orphan\w+Annie\b/, result)
```

Many test rigs come with an assertion for **Regular Expressions** on strings. The above checks:

- "Sandy" comes between word `\b`oundaries,
- anything comes between: `.*`
- "Little Orphan Annie" comes after,
  - she similarly has word `\b`oundaries, and
  - she may have any kind of one or more blank spaces (`\w+`) between the elements of her name.

Many unit test frameworks support regular expression matches. Page 368, for example, reveals a regular expression match in Perl.

Web e-search engines provide even fuzzier matches; they skip punctuation between parts of strings, permit words out of order, etc. They err on the side of returning pages. If assertions need *that* much fuzziness, they might instead need a better strategy.

Parsed Fuzzy Matches

Markup languages often require matches following the markup language's grammar, so the match forgives the same variations as the language forgives. HTML renderers, for example, dismiss all excess blanks, so we don't need assertions that fail when those irrelevant blanks change. We will soon learn forgiveness is a mixed blessing.

The examples here use HTML because it's widely familiar, but these principles apply to any scripting system.

A parsed fuzzy match on HTML containing "Sandy" and "Little Orphan Annie" could selectively only match those words in clear text, not inside HTML <tags>. Or vice-versa.

These match techniques decouple the important part of output from the transient part. For example, this Regular Expression Match might ensure a Web browser (wrapped by the object `@ie`) has connected with a server:

```
assert_no_match(
        /\<TITLE>Cannot find server\<\/TITLE>/,
        @ie.htmlNode.innerHTML
        )
```

(Browsers provide better ways to test their current state. That sample illustrates testing the GUI Layer, using terms introduced so far.)

Note the special escapes `\` before the `<TITLE>` tag to prevent the regular expression parser from interpreting the angle brackets, `<` and `>`, as commands. This is not yet a Parsed Fuzzy Match, because the matcher does not parse HTML—it parses regular expressions.

To truly match this situation, our tests need a library that parses HTML. A library that parses XML could also work, but only if the HTML were perfectly well-formed XHTML. XML parsers provide a high-level query system called XPath, to traverse XML nodes, match patterns, and return result sets.

Given XHTML containing "`<HEAD><TITLE>yack yack yack</TITLE></HEAD>`", the XPath address of that crucial information, "`yack yack yack`", is "`/HEAD/TITLE`". XML parsers convert XML notation into object models, and XPath queries nodes out of them, returning high-level objects. One of their `text` members contains "`yack yack yack`".

Let's try to use XPath to detect if Internet Explorer found its target server:

```
doc = Document.new(@ie.htmlNode.innerHTML)
e = XPath.first(doc, '/HEAD/TITLE')
assert_not_equal('Cannot find server', e.text)
```

That could have been the ideal way to extract the contents of the `<TITLE>` tag. But (as we will often discover when we probe an environment for testability) it does not work. Some HTML is also well-formed XHTML, but `@ie` controls Internet Explorer, whose built-in "Cannot find server" page is not well-formed XHTML. The parsing fails on the `Document.new()` statement.

Our own HTML should be the highest-quality XHTML. That expands the number of parsers we can use to test it with parsed fuzzy matches, and expands the number of ways to operate those parsers. The act of parsing also tests.

HTML has a strict mode—XHTML. But, in general, we will always encounter libraries that depend on forgiveness in ways that interfere with our testage. Different Web browsers forgive in different ways. We must remain on a higher level.

Tests that force code to exist will constrain more things than such low-level tests retrofitted to existing code could ever hope to constrain. And all such tests have a significant risk of hyperactivity. The Test-First Programming rules permit an easy Undo over a hard investigation into most unexpected test failures.

The user might not care if a given stretch of HTML said:

```
<strong><em>so what?</strong></em>
```

or:

```
<strong><em>so what?</em></strong>
```

Honestly, we don't care either. But our tests will, because the second is strictly well-formed. Web browsers forgive to enable Web development by hook or by crook—or by Notepad. That accelerates some kinds of development, but not ours. Our tests accelerate us, often by remaining more strict than humanly possible.

Page 377 drives these techniques to their illogical conclusion, a Bulk Parsed Fuzzy Match.

All these kinds of test cases provide a platform to "tune" their fuzziness, and to improve test accuracy as you develop.

Conclusion

GUIs present unique problems that will submit to many broadside attacks and many surgical interventions. Tests reduce risk, and right-sizing our effort reduces it more.

To apply these techniques to a software engineering project, we next learn where to place the techniques within the greater project lifecycle. We must adapt the solutions to the way most projects work—with colleagues, bosses, deadlines, and diverse user populations.

Chapter 4: *The Development Lifecycle with GUIs*

The previous chapters present some solutions in search of problems. Don't apply the solutions until you have their problems. This chapter explains the relationship between our techniques and complete projects.

When to Design the GUI

Naïve programmers often confuse the beginning of a project with the act of painting many controls on a form. "Should it look like this?" Their clients recognize empty controls, with suggestive names, as plans for feature sets with those names. Then programmers write response code into the event handlers "behind" these controls. Then they add all the business logic into the handlers. That path leads rapidly to spaghetti code with many problems. The most serious is logical code coupled to the GUI. Programmers cannot create Logic Layer objects alone without their GUI.

Form painters can create instant, fascinating, and colorful stage-prop demonstrations, with nothing inside them, tempting clients to pay for more. If you write such a project, to assist planning, keep its source out of your codebase. After your first iteration, real features will have a real GUI. It may present fewer controls, so demonstrate it to those customers, as soon as possible, to manage their expectations. Lead them to drive the minimal but bug-free features, and experiment. Reinforce to them your team measures progress only by completed features, not by completed but disconnected modules, or by useless GUI controls.

Development follows two nested cycles. Writing source code to implement features is the inner cycle, and delivering versions to users is the outer cycle. In each delivery, all real features have controls, and all controls have real features.

Some Agile literature speaks of "GUI-last" programming. When business analysts request feature sets, they censor all usability assumptions. Then programmers write their logic without any user-friendly interface at all. Only after it works do they risk painting a quick, Spartan user interface over its minimal functions. This gives the Customer Team a reference point from which to drive the growth of a project's usability.

Although "GUI-first" and GUI-last both distort the big picture, GUI-last illustrates many positive benefits. Encapsulation is hierarchical, and the Logic Layer should not couple with the GUI Layer.

"Encapsulation is hierarchical" means one module uses very few elements of another module—the ones that *want* to be used. A Logic Layer should only use (and be used by) very few elements of a Representation Layer. From the logic's point of view, the Representation Layer is a Façade to hide the computer's most important and most squirrelly peripheral—the user.

In a mature project, one can always add a new test onto the Logic Layer to express some feature there, without the need to run any GUI to access that feature. That is the meaning of the "GUI-last" guideline—the layers decouple, and the Logic and Representation Layers can compile and execute "headless", without the humble GUI Layer. As a project matures, all its elements must upgrade (and refactor) together. None may get too far ahead of others.

A program's usability needn't change as often as its internals.

No matter how usable your GUI, users must make an effort to learn to extract value from it. They need to preserve that learning. Tests should help programmers rapidly upgrade without allowing GUI changes that could interrupt users' flow. And tests will help construct new, elaborate features without changing the usability of older features.

In a mature project, the user may legitimately experience a long "click path" before arriving at a given window in a given state. For example, typing in and editing a new 474-page document in a word processor, full of unique & challenging spelling and grammar issues, should take some time. However, new test cases must be easy to write, no matter what state they place their GUI objects into. Put another way, the GUI under test decouples its state from its click path.

A test case might place a GUI into a state the user cannot drive to. Don't use that technique to conceal low velocities developing a GUI's support features. Page 155, among others, illustrates that technique concealing a low velocity.

Begin your GUI work when your team requests it, then ensure the layers decouple. Both yourself and your colleagues should work on the business logic without any GUI entanglements. All business logic should execute (through tests, at least) in complete isolation from any GUI. Ensure your colleagues can use (and can write tests for) your GUI. To learn if your GUI is useful, make sure its features help them manually experiment with the Logic Layer.

Do not deliver the first version of the program, to any significant user population, without formally and thoroughly cleaning and reviewing the current GUI's usability and esthetics.

Big GUI Design Up Front

Mike Cohn's book, *User Stories Applied: For Agile Software Development*, explains why GUI usability should not change freely, in a sub-chapter called *User Stories and the User Interface*, citing Larry Constantine & Lucy Lockwood's prolific research into "Usage-Centered Design".

GUI appearances refactor very easily; form painters simply drag and drop controls. (We wish all source code refactored this easily.) Teams frequently change their own tools' GUIs, so they may feel that any change to a production GUI, whether minor or radical, must be an improvement that should delight users. But those authors present the opinion that the same program with a different user interface might cause stress. Users memorized click paths and appearances, bonding with the early versions that boosted their productivity.

Leverage that bonding. No programmer can guess which GUI "improvement" will damage a user's experience. Refactoring users is hard (quoting one Martha Lindeman), so usability requires more up-front research, requirement analysis, and proactive planning than other aspects of Agile development. Usability analysis requires a third feedback cycle, outside the nested inner and outer cycles of implementation and delivery.

Changing a GUI might split a user population between early adopters and new converts. Some populations have different needs, compelling a split. All these user populations deserve frequent deliveries, so projects may need more technical support, to keep everyone in the loop.

Version with Skins

Sometimes two or more different user interfaces must cover the same Logic Layer. Each could be very similar or completely different. The GUI or Representation Layer might change.

One common industry reason is versioning. Version 5, for example, will have many new features, and these deserve re-designed usability concepts. Version 4, however, shares the older features in the same Logic Layer as V5 uses. Users awaiting V4 upgrades should not be told, "Wait for V5. It has the feature you requested, but you must wait for us to give it a new and improved user interface that looks and feels completely different!"

Forking the entire codebase is a common industry response. That technique compounds incredible problems, without even short-term benefits. A team either neglects its V4 users, or mirrors every upgrade between V4 and V5. Forking a codebase is duplication, like any other. Left unfixed, it drags down a team's productivity.

As soon as the V4.1 user interface no longer "looks like" V4, diverge only those aspects of the GUI which differ. The program's skin will be a configuration setting—not a product of "Conditional Compilation", or a separate codebase. Methodologies that prevent GUI code from turning into spaghetti will easily support forking a view. Multiple skins can force their duplicated code to converge into their Representation and Logic Layers, keeping these honest.

Many projects release more than one application built out of a single core. The differences between each "flavor" of application could be as simple as tiered price/feature points—Personal, Standard, Premium, Enterprise Editions—or as complex as different user interfaces tuned to match different hardware assemblies. The Software Engineering Institute calls this topic "Software Product Lines", following the work of Linda Northrop and Paul Clements. A GUI should also address it with skins. Tests must cover all skins, flavors, and locales when any of them change.

All projects developed by Extreme Programming have one more skin. Acceptance Test Frameworks form a completely different skin on a Logic Layer, permitting Customer Team users to rapidly author new tests, bypassing the GUI designed for end-users.

Later in this book, the *Broadband Feedback* Principle will lead to an Acceptance Test Framework that captures animations of all skins. They permit the Customer Team to review appearances and scenarios faster than they can install and operate all the different versions, locales, and flavors that a successful application might spawn.

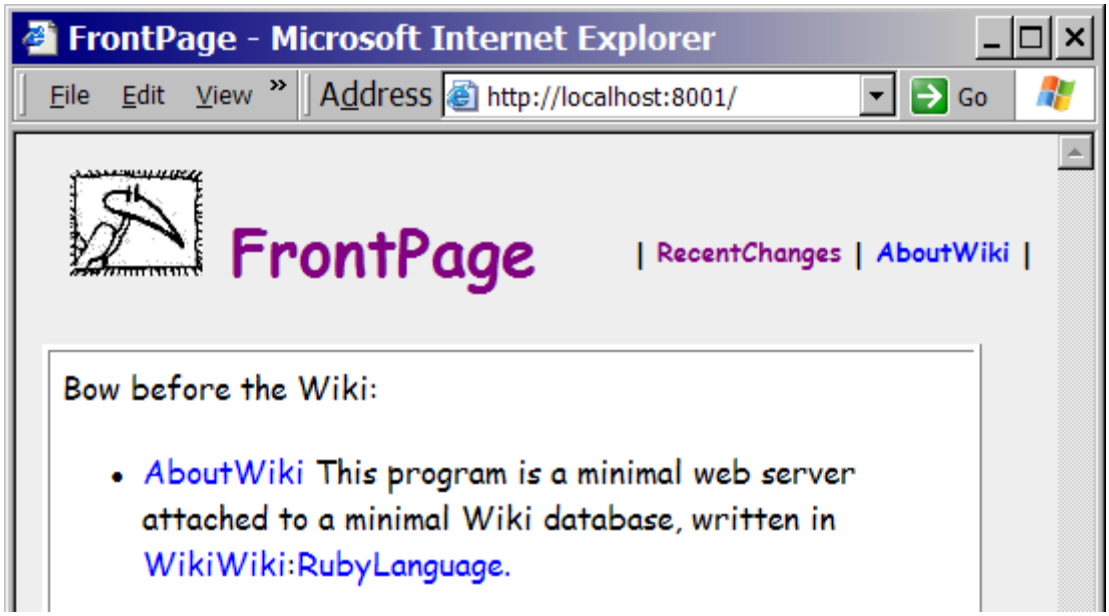
Skins should all pass a common set of test cases. The Abstract Tests pattern reuses test cases across multiple conditions, such as skins.

Page 264 presents Abstract Tests constraining different skins, one per locale.

Abstract Tests

The Web Case Study, on page 365, features MiniRubyWiki (MRW), but only covers a few episodes of its development. It uses a variation of the practice "Version with Skins", so its tests constrain its behavior under two different kinds of servers.

Page 375 defines "Wiki Wiki"; for now recognize it as a dynamic web site that looks like this:



When you click the big "FrontPage" link, the server searches for that string in its Web site, and returns an XHTML page containing links to all pages containing the text "FrontPage".

The module `miniServer.rb` implements a minimal Web server based on a raw `socket` connection. It pulls strings directly out of a TCP/IP connection and parses their variables, obeying only the most important details of the HTTP recommendations. This efficiently bypasses many features that "real" Web servers enforce. `miniServer.rb` can serve on alternate ports, such as `8001`, to peacefully coexist with other servers.

Another skin, `wiki.rb`, bonds with real Web servers via the traditional Common Gateway Interface. I test that with Apache `httpd`, serving on the default HTTP port, `80`.

Both servers front the same Wiki features, so MRW's test rig contains a suite of test cases that run twice, following the Abstract Template Pattern.

The base test suite wraps Internet Explorer, to fetch and interpret pages. All tests follow the same general pattern. They `surf()` to a specific URL, return a page, find some text on it, and compare that text to a reference:

```
    class TestViaInternetExplorer < Test::Unit::TestCase
...
        def test_defaultToFrontPage()
            surf('FrontPage')

            regexp =
    /href="#{getPrefix()}SearchPage(&amp;|\?)search=FrontPage"/

            assert_match(regexp, getHTML())
        end
```

```
    end
```

That case reuses test fixtures, so it's short, inscrutable, and only obvious if you know that `@ie` is an unseen member variable referring to an instance of Internet Explorer, and `surf()` navigates it to a Wiki page.

`surf()` wears two concrete skins:

```
    class TestMiniServer < TestViaInternetExplorer
        def getPrefix()
                return ''
            end
…
        def surf(page)
            href = "http://localhost:8001/#{page}"
            navigate(href)
        end
…
    end

    class TestApache < TestViaInternetExplorer
…
        def getPrefix()
            return 'wiki.rb\?'
        end

        def surf(page)
            href = "http://localhost/wiki.rb?#{page}"
            navigate(href)
        end
…
    end
```

`TestMiniServer#surf()` heads for `http://localhost:8001/`, and `TestApache#surf()` hits `http://localhost/wiki.rb?`. The example test ensures both `miniServer.rb` and `wiki.rb` return a page with a hyperlink to the search system containing "`FrontPage`".

The fulcrum of the test is the overridden method `getPrefix()`. It returns either an empty string or `'wiki.rb\?'`. When `getHTML()` queries IE for the HTML contents of its page, our Regular Expression Match then determines that our page has a link on it that searches all Wiki pages for "`FrontPage`". That's the FrontPage's title's behavior when you click on it, so only `miniServer.rb`'s FrontPage will contain this:

```
<a href="SearchPage&amp;search=FrontPage" title=
"Click to search for this string">FrontPage</a>
```

But the Apache's FrontPage contains a longer `href` value:

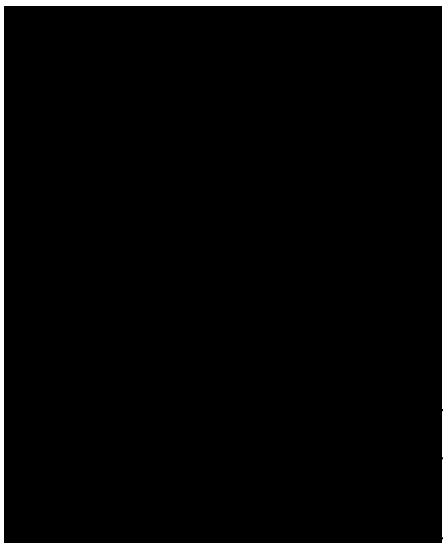
```
<a href="wiki.rb?SearchPage&amp;search=FrontPage" title=
"Click to search for this string">FrontPage</a>
```

To detect that difference, the test's `assert_match()` calls `getPrefix()`, nestled inside its Regular Expression Match:

```
        regexp =
/href="#{getPrefix()}SearchPage(&amp;|\?)search=FrontPage"/

        assert_match(regexp, getHTML())
```

Ruby embeds executable statements inside strings or regular expressions, escaped with `#{}`, so `#{getPrefix()}` introduces different search string details for each skins.

This test program's `main()` function adds `TestMiniServer.suite()` and `TestApache.suite()` to its list of test cases, so both concrete suites express the same shared tests, including `test_defaultToFrontPage()`:



A TestRunner kicks off the game by calling `Test::Unit::TestCase::suite()`, to fetch a global list of cases. For each case, it calls `setUp()`, the case's `test_*` method, and `tearDown()`. `setUp()` uses virtual dispatch to call a concrete implementation of `surf()`, in one of the derived classes. Then `test_defaultToFrontPage()` calls a concrete implementation of `getPrefix()`, again in one of the derived classes.

That sequence diagram illustrates Contractive Delegation. `TestRunner` and `TestCase` are completely generic. `TestMiniServer` and `TestApache` are specific, and `TestViaInternet-Explorer` is partially generic. Method calls going to the right become more generic. The parameters to those methods are specific—they contract their delegates. Method calls to the left dispatch virtually into concrete test classes, and fetch data out for those parameters.

Sharing a test suite between many skins (or versions or product lines) helps engineers think about only one skin without accidentally breaking another one. (And note that naïvely invoking a *Temporary Visual Inspection* from `test_defaultToFrontPage()` would raise two web browsers, each containing the same page!)

With a little practice (and our Principles), GUI tests might become *too* easy. Programmers might use them to test-first new Logic Layer features. These excessive tests might cause some problems which refactoring was invented to solve.

Smart UI AntiPattern

Eric Evans's influential tome, *Domain Driven Design: Tackling Complexity in the Heart of Software*, enforces a correspondence between entities a domain expert would recognize and objects that satisfy programmers' design principles. If a domain expert pontificates, "Real numbers shall inherit integers," or vice versa, programmers shall not cringe but rejoice. They will write classes labeled according to matching domain entities, and ensure their program appears to contain these entities. The internal inheritance arrangements may differ.

Just as the book *TFUI* recommends when not to test-first a lowly esthetic detail, the book *DDD* recommends when to bypass careful architecting, and fulfill a short iteration's requirements using good old-fashioned "Rapid Application Development". Eric recommends, for some projects, to paint a form, populate it with controls, write business logic directly into their event handlers, deliver the result, and immediately boost user productivity.

This strategy's short-term benefits are plentiful. Form painters, tutorials, and wizards redoundingly support the effort. When the same vendor supplies the GUI Toolkit and database, they often bind seamlessly. The resulting design resembles one big happily coupling cluster.

Sadly, tragically, TFUI Principles reinforce this AntiPattern. When test fixtures provide rapid manual review of forms, they might also rapidly review database contents. That strategy might lead to tests that drive a user interface, submit data into its event handlers, and query the correct results in a database. GUI tests might drive Logic Layer development. One batch of tests might simultaneously and aggressively reinforce all those latent layers.

The effect works similar to a common misconception regarding "Acceptance Tests". Folks sometimes use that term to mean "tests that drive a GUI". The misconception deepens when those GUI tests devolve into "tests that drive a GUI to exercise Logic Layer features".

Do such tests constrain the GUI Layer? or the Logic Layer?

"Capture/Playback" test rigs often lead to that misconception. When such tests reach through a GUI to constrain business logic, those tests pin down GUI Layer aspects, inhibiting new features. Changing such a GUI breaks acceptance tests, which then report failures in your Logic Layer!

If misdirected acceptance tests unfairly constrained a legacy project's GUI, consider going back a few pages, and applying the practice Version with Skins. Leave the legacy skin in place, to support these tests, while growing a new skin to support new features.

If you begin your project using the Smart UI AntiPattern, Test-First Programming will penetrate your latent Logic Layer. Refactors that merge common behaviors between event handlers can produce new functions, which can then divest themselves of GUI Toolkit identifiers. New tests, following the Child Test pattern, will address these functions directly, and bypass the GUI-oriented fixtures. Eventually, a healthy layered structure might grow.

Test-first changes which strategies are always AntiPatterns, and which strategies can lead such latent designs into expression. A little refactoring can easily nudge things in the right direction.

Sane Subset

This book presents code that follows many technical guidelines. Not all of them are documented. (And not all of them are ideal!) Good guidelines keep source code within a "comfort zone". Programming languages provide extraordinarily wide design spaces, much wider than hardware designs enjoy, with many tricks and backdoors that could provide hours of pleasant diversion writing obfuscated code.

Don't write like that on the job. Always write similar, obvious statements to do similar, simple things. Engineers learn their working vocabulary from experience and teammates. A healthy subset of all possible language constructions keeps everyone sane.

Programming tutorials often contain recommendations like:

- "Make all data members private to their classes."
- "Program to the interface, not the implementation."
- "No global variables."
- "High level policy should not depend on low level detail."
- "Declare a separate interface for each category of client to a class."
- "No down-casting. Don't use an object's type in a conditional."
- "Don't do too much on one line."
- "Use `return` to prematurely exit nested loops."
- "Use complete, pronounceable names that reveal intent."
- "Dependencies between modules should not form cycles."
- "Don't allow client objects to tell the difference between servant objects using the same interface."
- "Don't use `anObject.getSomething().getSomethingElse()...` too deeply—tell the first object what your high-level intent is, and let it pass the message."
- "Different variables holding the same datum should have the same name."
- "Never return null. Never accept null as a parameter."
- "Seek ways to make virtual methods private"
- "In C++, don't abuse `#define`."
- "Put a class's `public` things at the top; `private` things below."
- "Give identifiers the narrowest scope possible."
- Etc…

Such rules are hardly exalted wisdom or exact engineering constraints (and neither is "test-first"). They are easy to remember, and nearly always preserve code flexibility. Rules are meant to be made, followed, and broken.

(Those curious how to follow the rule "never return null" may research a Refactor called "Introduce Null Object".)

The least sane language in this book, C++, provides more than enough rope to shoot yourself in the foot. Many fine books describe its numerous "don't" rules; some influential ones are:

- *C++ Coding Standards: Rules, Guidelines, and Best Practices* by Herb Sutter and Andrei Alexandrescu
- *The C++ Programming Language 3rd Edition* by Bjarne Stroustrup
- the *Effective C++* books by Scott Meyers
- the *Exceptional C++* books by Herb Sutter
- *Large Scale C++ Software Design* by John Lakos
- *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples* by Barton & Nackman.

The average C++ code in this industry fails to live up to those simple guidelines. Page 186 lists a few more.

Tests Change your Sane Subset

With fewer tests, compilers must check more things for us, such as whether strangers attempt to access our private state, or whether object references passed into method parameters are the correct type. With more tests, the risk that a "mistake" with state or types could "accidentally work" becomes less risky. Prevent minor breaches from growing major, but don't micromanage a design.

Your GUI Toolkit comes bundled with support applications that may generate code. These bring their own style rules. Some form painters create form objects with only `private` control objects, and some declare all event handlers as `private`. They hint that forms are also objects, and other objects must only command them by calling their public methods. Other objects should not reach inside a form object, grab one of its controls, Get data from it, process that data, and Set it back. Form objects, like all other objects, should collaborate with their peers with useful public methods. A form's peers should not *Simulate User Input*, and "click on" its controls. Form painters create `private` data members, control members, and event handlers to remind programmers to encapsulate behavior.

If users can click on controls, tests should be allowed to click on them too.

Some GUI Toolkits can "reflect" controls by name from a list. Leaving controls `private`, then fetching them by name, is not really private. It's like a speakeasy or rave that allows guests to enter with a password. Extra enforcements don't guarantee a healthy design (just as extra bouncers don't guarantee a healthy rave). To simplify testing, make control members public, and refactor production code to resist the temptation to abuse them.

Continuous code review, and **Programming by Intention**, will help forms grow useful interfaces. No peers will feel the need to access *de-facto* public members, or sneak in with a password.

While this book emphasizes that tests are a developer's user interface on a project, tests do not follow many usability rules. For example, when production code creates a secret temporary file, it might use a method like `GetTempFileName()`. That returns a path to a deeply hidden folder, unique to the current user, and a cryptic file name. Tests should not use this system to create temporary files. Developers often need to rapidly find and read their tests' scratch files. Test cases should throw them into an obvious, common folder, such as `C:/TEMP`, and give them obvious names.

Tests also push many common design notions over the sanity border. For example, many traditional forms' constructors take a handle to a database, or to a Representation Layer object. Test cases must construct such forms to test their controls. If a target control's event handlers don't send messages to that Representation Layer object, the test case should not need to create it. The form's constructor must not demand this object.

An expensive `setUp()` fixture is a Design Smell.

A test case should not construct objects—even Mock Objects—that a test case will not command its testee to use. This analysis suggests the form should take its Representation Layer object in a Set method, or its constructor should accept `NULL`. That violates the common guideline that constructors should prepare an object for any of its production behaviors. The underlying guideline requires an object's constructor to do the minimum required to prevent the next method from crashing.

Refactoring also changes your Sane Subset. Some published style guidelines advise carefully selecting and standardizing your abbreviations. If you instead enforce complete names

(and a few common acronyms), then statements such as "`putSvgIntoCanvas(aCanvas)`" betray their duplicated concepts. Complete names can expose duplicated ideas, leading to refactors that merge them: "`aCanvas.addSvg()`".

Continuous Integration

As a codebase grows, teams frequently submit working source code to a database called a version control system. Every time source becomes sufficiently better that others should use it, even indirectly, it should merge with the version controller's database.

Some version controllers force developers to acquire exclusive locks on files before changing them. Configure your version controller to leave all files checked out all the time. No engineer should hesitate, or interrupt their Flow, before changing a file.

When the code improves enough for others to use, integrate following these steps:

1. run all tests
2. merge your source with the most recent integrated version
3. run all tests
4. commit your recent changes into the database
5. run all tests
6. announce you have checked in a new change.

If any step requires excessive keystrokes or mouse abuse, write scripts to automate environmental manipulations. Only novices need GUIs; they are training wheels for tools. Engineers take them off and ride.

Some teams require members to declare Step 1, using a mutually exclusive semaphore. Before checking in a change, engineers must move to an integration workstation, or obtain a unique token, such as an Official Rubber Integration Chicken.

Steps 1, 3, and 5 share the same script. If it takes too long to run, seek and remove performance bottlenecks.

If Step 1 fails, you are not ready to integrate.

If the same source files change at different workstations, most version control systems, at Step 2, will merge the changes together. If the same line changes on two workstations, this merge system breaks, and the version control system demands manual intervention.

If Steps 1, 2 or 3 fail, you have a choice:

- debug the failing test
- erase your own changes, get a fresh copy, and start your changes again.

No other Agile practice advises debugging a failing test, so now is not a good time to start. If the problem isn't simple and obvious, then the choice "erase your recent changes" sounds a little counterproductive. Minimizing exposure to this loss is more productive than the alternatives.

The longer you delay each integration, the higher all integrations cost.

Small frequent integrations always accumulate less cost than large sporadic integrations. Delayed integration increases risks of conflicts, broken builds, or failing tests, at Step 3.

Delayed integration raises the cost of throwing your local changes away, or even of copying them out to a scratch folder.

The more often a team integrates, the less likely become failures at Step 3. Each small change that leaves code demonstrably better deserves a check-in. Engineers should leverage the ability to play with source, attempt some experimental changes, and then discard them by erasing local files and downloading fresh copies. Frequently committing valid versions creates checkpoints that future bulk Undo operations can return to.

Test-First Programming enables personal Flow by reinforcing a mental model of a module's state. Continuous Integration enables team Flow by reinforcing a collective mental model of a project's state. So failing tests at Step 3 are a Good Thing—they efficiently warned you to stop developing, and to reevaluate your mental model. If the code only changed a little, the problem might be obvious enough to fix right away.

You always have the option to erase your own changes, get a fresh copy of the project source, and make your changes again. You should have only a few changes to make, they will be easier this next time around, and you will remember to integrate them more often.

To help everyone remember, automate Step 6. Many teams use build script systems (such as Java's ANT) that can play a sound at integration time. Any build script could call a command line program to play a sound, too. If everyone sits together (and doesn't listen to music on earphones), they will hear your integration event. They know their changes now live on borrowed time, so they will seek the next opportunity to integrate, too. And you will hear their sounds.

Contraindicating Integration

Not all code changes deserve instant immortality. Some code changes might contain embarrassing GUI experiments that not all reviewers would catch. Programmers must quarantine their workstations until finishing their experiments, and then clean their effects out of the code.

Never fork a codebase to create a Production Build. Fork a codebase to experiment, then erase the fork, and apply what you learned to the real codebase. Don't integrate aggressive experiments.

Page 256 demonstrates this effect. We change a project experimentally, producing a display bug that only people fluent in a complex Asian language would recognize. If we integrated, then forgot to redact the experiment, not all of our colleagues could catch the error. In my experience, healthy software engineering lifecycles rely on forgetfulness.

Don't integrate "Spike Solutions". Normal development depends on many minor experiments. These must chronically remain ready for integration.

Don't Leave reveal() Turned on

When you activate a *Temporary Visual Inspection* or *Interactive Test*, you need to forget integration. When you integrate, you need to forget any temporary windows you left turned on. Pass your `USERNAME` into the temporary window fixture, such as `revealFor("phlip")`. That fixture should match this name to the `USERNAME` environmental variable to trigger a display. The Case Study NanoCppUnit demonstrates this technique, on page 199.

An aggressive coding session, while adding a feature, might strew `revealFor()` calls across many modules. Write your name there to permit integration as often as the code improves; more often than you could seek and remove all the `revealFor()` calls.

Resist the temptation to write someone else's name there.

Flow

This is the goal of the TFUI Principles, and the goal of the software development effort around them. In this zone, every thought connects. When an individual emerges from Flow, they may be surprised to find hours have passed. The sun may have set or risen; they didn't notice.

The rhythm of testing propels Flow, as your short-term memory maintains an image of your current task. This memory behaves like a CPU's storage cache. Interruptions that replace the cache damage Flow. Peaks in your Flow's intensity often trigger the exact moment your loved ones, even from long distances, decide to interrupt you.

Our technical goals—*Query Visual Appearance*, *Simulate User Input*, etc.—have distinct definitions, so we know unambiguously when we have them. Our real goal is Flow—a subjective, atmospheric experience. This book's Part III tells the tales of Sluggo and Nancy (in Chapter 14: *Sluggo at Work* on 429, and Chapter 15: *Nancy at Work* on 434), using Code-and-Fix and Test-Driven Development, respectively. Both use Flow. But Sluggo uses it to ignore a high bug rate while he drives a debugger around the code. Nancy uses the Flow of testing to relentlessly prevent bugs.

Between test runs, only perform so few edits that all of them fit in your short-term memory. Sluggo uses Flow to memorize sequences of dozens of GUI manipulations, to avoid the bugs he keeps secret. While making each sloppy, fragile change, he walks on safe paths through his minefield.

When you test, between hitting the test button and receiving a Green Bar, your test cases must take care of every possible detail that enables your project's success; more details than fit in short-term memory. The Green Bar, like a casino slot machine, stimulates your brain's pleasure center, except that test-first nearly always wins. This stimulation propels Flow, and reinforces your mental model.

To ensure tests cover deeply, constantly try to break things. Do not memorize sequences of dozens of GUI manipulations to avoid. Stomp on the mines. (The book *How to Break Software: A Practical Guide to Testing*, by James A. Whittaker, gives excellent, detailed advice for spotting the tell-tale lumps in the ground.)

Hit the test button before starting a session. Hit it before and after taking a break, or servicing an interruption. Naturally, hit it before and after integrating. Hit it when changing pairs. Hit it before any change in direction. Hit it before and after upgrading a library. Hit it before and after snack time. Hit it if you can't remember if you just hit it.

But the Bar's stimulating color might not always be Green. Some new assertions must fail, forcing code changes. To test our tests, in the negative, we write such a new assertion, then hit the test button. When adding the code, we expect the test to pass, and when refactoring we expect each small change to pass.

Our expectations channel intent into the code, and laminate its defenses. To reinforce, each time you hit the test button think the following, or say it aloud to your programming pair:

- **New structure**—"Predict compiler error."
- **New assertion**—"Predict failure."
- **New code**—"Predict success."
- **Refactoring step**—"Predict success."

When writing an assertion and ensuring it fails for the correct reason, you predict a Red Bar, and getting one stimulates Flow.

GUIs require occasional visual checks, so predict their results too.

Continuous Integration pumps the cycle of the team's Flow. You might make a change that I don't yet understand, so I trust you tested it, and I know my recent changes are simple enough to merge. When you integrate, I integrate very soon. Agile teams expect a certain range of technical details to always perform smoothly, so any deviations receive instant notice. Teams maintain their Flow as the ultimate metric of project health. When these cycles break down, the entire team should work to repair them.

Time vs. *Effort*

Agile development leverages short, predictable cycles. When some tasks appear long or unpredictable, teams notice. New territory causes unpredictability. Writing lots of code without any tests takes a team into new territory. Some risks are less avoidable.

When a team decides to add a legacy library to their project, it may come without tests, and without a design that promotes testing. Teams must rate the cost of adopting such a library against that of rewriting one from scratch. (And TFP makes the latter less risky than traditional techniques.)

This book discusses legacy GUI Toolkits that come without tests, and with incomplete hooks and adapters to enable testing.

All serious applications need GUIs, to rapidly teach users to enter complex commands. But adding a new kind of GUI to an application generates the risk of unpredictable delays. This

book recommends extra research into corners of GUI architecture that might be a little dark. That trades a lot of risk debugging for a little risk researching unsupportable features. Each new kinds of GUI require more research, so each causes unpredictable delays researching how to defeat their risks.

Meanwhile, GUIs come bundled with tempting form painters, wizards, and debuggers, all carefully designed to lead novice programmers directly into the hard features. Starting a new GUI feature using a wizard instead of a test is very tempting. So I must make a case, again, for how important TFP is to the rest of the project's schedule. The up-front investment will pay for itself many times over:

The Code-and-Fix line starts easy, while you use a GUI Toolkit's bundled gadgets, such as Wizards and form painters. But as you run out of wiggle room to rapidly change code without causing obscure bugs (and as you go where that Wizard can't follow), that line trends up into unsustainable regions. Once there, each new effort adds bugs, and these take time to search and destroy. When they are fixed, you have the choice between letting design quality slide, or improving it at the risk of adding more bugs. This forms a vicious cycle: Low design quality increases the odds that new features *and* design upgrades generate new bugs.

Everyone has experienced codebases, maintained for a few years, that became too hard to change, and were thrown away. For example, Microsoft replaced `MSDev.exe` (Visual Studio 6) with `DevEnv.exe` (Visual Studio 7), instead of upgrading. The excuse "but `MSDev.exe` is very big, and `DevEnv.exe` is very different" should not matter.

The high-effort areas of each kind of project do not just take a lot of time. They interfere with tracking and estimating. The higher a point on the chart, the wider the spread between estimates and actual times, and the higher risks of delays to research into mysteries and to concoct patches & compromises. High-risk activities cause cruft, because refactoring their excess code away similarly adds risk. Risks cause stress.

Your corporate culture should not reward heroism, or volunteering to experience risk.

Heroism is not sustainable.

When bosses order their subordinates to volunteer, they destroy corporate hopes. Teams must be empowered to seek ways to reduce risks. If tests do not yet control a project, the benefit of installing them always outweighs the cost. Adding tests to a test resistant situation takes time, temporarily increases risk, and requires a complete team effort.

After you cross the peak and enter the low-effort phase, your environment permits easy Flow with minimal distractions. Expect to encounter such a peak, and schedule time to cross it, for each new library. Software lifecycle books call this the "Exploration Phase", or an "Architectural Spike Solution".

If a new library troubles you, determine if it needs exploration. Convert its sample code into a Learner Test, then see how easily that test starts answering your questions. We'll call the exploration phase for a specific library "**Bootstrapping**", before we learn to write the first relevant test for a given situation.

Down in your own coal mine, tests are canaries whose nervousness shows the presence of evil design vapors. However, when entering someone else's coal mine—a library you must construct tests around—expect to expend many nervous canaries!

Your occupational behavior may raise naïve concerns here. Suppose the program needs to Get a complex variable, but you first spend a lot of time learning to Set that variable—so a test case can transmit sample data into the tested function, of course. The potential benefits might not be apparent.

If it does not have a test (or a visual check), it adds risk.

Adding features without their tests creates "artificial velocity" that defers the hard part of programming until later, after it grows much harder. Adding tests and features without refactoring creates artificial velocity, too. Developing a GUI Layer without writing test fixtures that enable our TFUI Principles … might work, or might not.

Conclusion

This book offers simple goals that many GUI Toolkits will fight to the death (theirs, of course). Understand the priorities, and push most to enable Flow. Some Principles listed here, for some platforms, are impossible. Some Principles we can let slide, so long as they slide in the right direction:

A project grows test fixtures tuned to enable the TFUI Principles for the intersection of the project's features and its GUI Toolkit.

Eventually programmers can change features rapidly for long spells without spending excessive time displaying the GUI, or hunting bugs. Time estimates become short and accurate.

A mature TFP session is pleasant and relaxing in practice, but describing it generates tedious narration without challenges. After the Hobbits reach Rivendell, Tolkien's stories skip ahead a month or so—into the Misty Mountains. After development becomes steady and obvious, this book's Case Studies will leave the project. Our adventures start on the far left of the *Time* vs. *Effort* chart, and climb the passes between the peaks.

Part II: Case Studies

Humans learn by example. After reading Part I, don't write a GUI using test-first yet. That Part lacks complete source code samples.

Part I arranges topics by theme. Real projects must arrange topics in the order that engineers perform tasks within an iteration. Engineers seek ways to cause the next problem they intend to solve. This strategy mixes all the Part I topics together, out of thematic order.

Serious, useful projects force real development issues to the surface. Then we discuss the problems, using terms defined in Part I for efficiency. Don't skip Part I, then attempt to interpret the jargon or emulate one of these sample projects. Read Part I, then skip to the Case Study that uses your favorite system, or that illustrates a fix for your stickiest problem.

Alternately, go with the Flow, and read the Case Studies in order. Each defines the language and library systems it uses, and starts simple. The first two Case Studies, SVG Canvas and Family Tree, use downloadable systems, reveal each refactoring step, and achieve a high bang/buck ratio.

Chapter 17: *Exercises*, on page 459, lists over a hundred engineering tasks to extend these Case Studies.

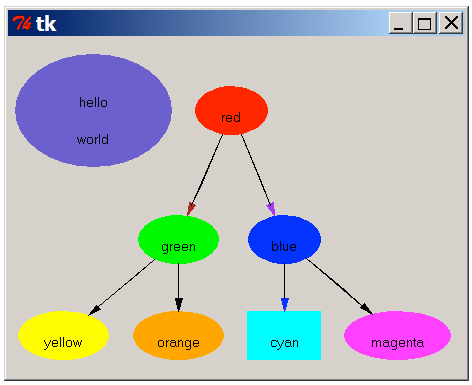
TODO: put these into the same format as the other tables of links to Case Studies or Chapters…
TODO: put a thumb-tab to this page, or put it inside the covers or something
TODO caps or upstyle in here, and add page numbers

Chapter 5: ***SVG Canvas***	A close-up, gentle introduction that builds a custom control that displays graph diagrams. Canvases, GraphViz, **Just Another Library**, Refactoring, Ruby, SVG, **Temporary Visual Inspections**, **Test-Driven Development**, Tk, XML, & XPath.
Chapter 6: ***Family Tree***	Extends the graph control to support user interactions. Canvases, Graphical Editing, GraphViz, Ruby, SVG, **Temporary Interactive Tests**, Tk, XML, & XPath.
Chapter 7: ***NanoCppUnit***	A foundation of ***Fault Navigation***, leading some of the hardest low-level GUI development issues toward Agility. COM, MS Windows, *Regulate the Event Queue*, Visual C++, WTL, XML, XPath.
Chapter 8: ***Model View Controller***	Using a simple data entry form to illustrate good architectures emerging from "First Principles". COM, ***Fault Navigation***, MS Windows, Regulate the Event Queue, Visual C++, WTL, XML, XPath.

Chapter 9: ***Broadband Feedback***	Extend the previous project into complex visual displays, low-level graphics, localization, animation, and Event-Driven Architectures, GDI, ImageMagick, Localization, MetaFiles, MS Windows, Progress Bars, **Query Visual Appearances**, Sanskrit, Skins, Unicode, Uniscribe, Visual C++, WTL, XML, XPath.
Chapter 10: ***Embedded GNU-C++ Mophun™ Games***	Embedded software and games bring their own rules. We start a small game to play on a cell phone. Animation, Emulators, Event-Driven Architectures, **Simulate User Input**.
Chapter 12: ***Fractal Life Engine***	Embedding a language can make very hard systems soft and easy to change. Query Visual Appearances via Mock Graphics, using Qt, Ruby, and a fractal interpreter named Flea.
Chapter 13: ***The Web***	A survey of techniques, and a Web-Enabled Acceptance Test Frameworks that that presents a gallery of images from the Broadband Feedback project. CGI, COM, DOM, HttpUnit, ImageMagick, Java, Konqueror, **One Button Testing**, Perl, Ruby, WebUnit, WebInject, XHTML, XML, XPath, and XSLT.

Chapter 5: *SVG Canvas*

Our first Case Study focuses on the Test-First Programming operation, and the Principles *Just Another Library* and *Query Visual Appearance*. We also introduce many systems, such as XPath, that show up everywhere else. This project creates a canvas that displays arbitrarily complex graphs of nodes, like this:



The next Case Study extends this project's tests to *Simulate User Input*.
Here's this chapter's sequence of activities:

To make things easiest, we will use freely available and highly portable tools—Ruby and TkCanvas. Other Case Studies address toolkits that make things harder.

Instructions how to obtain and use these tools appear shortly. The first three typically bundle together:

- Ruby
- Ruby/Tk
- Ruby/REXML
- GraphViz

Ruby supports a GUI Toolkit called Ruby/Tk, which wraps the legendary TCL/Tk toolkit. The Tk Canvas treats displayed graphics as objects, and Ruby's `TkCanvas` class provides subsidiary classes for each graphic element. `TkcText` represents an item of text, `TkcRectangle` represents a rectangle, etc. Each object has methods to Set and Get a long list of display properties. Changing these properties, in batches, permits the hosting Canvas object to efficiently repaint.

You might reproduce this project using the TkCanvas under a different language. The powerful and subtle Tk library has ported from its native TCL to many languages. This project avoids Ruby complexities, such as meta-classes, that other languages implement differently.

This Case Study leads participants to feel the Flow of a TFP session with GUIs, unencumbered by excessive research into the Principles *Just Another Library* or *Query Visual Appearances*. TkCanvas's rich programming environment rapidly enables both those effects.

Attempting this project with a different canvas will lead to unpredictable results. Learning Ruby by playing along with this Case Study will be easier than forcing other canvas controls to support TkCanvas's features.

Ruby in a Nutshell

In Ruby, everything is an object, including classes and integers. This leads to a curious effect. When writing Ruby code, we may forget as much about Ruby's technical prowess as we like. We never need to remember or accommodate Ruby's basic principle, which is that everything is an object. Ruby's basic principle stays out of your way.

Learn more about Ruby via *Programming Ruby: The Pragmatic Programmer's Guide* by David Thomas & Andrew Hunt.

Ruby comes with common Linuces, such as RedHat, and Steve Steiner, Curt Hibbs, and Andy Hunt maintain a full-featured Ruby installer for MS Windows, complete with Tk and hence TkCanvas, at `http://rubyforge.org/projects/rubyinstaller/`.

Ruby competes with both Perl and Smalltalk. I will avoid many shortcuts it provides, and will explain those few idioms that differ from the mainstream as we go. Ruby permits a range of expressiveness; we leave that knob set to "didactic".

On page 116, we write a useful function, `doc(anObject)`, which prints the name of all methods for any object.

TODO more citations, glue

Ruby distributions provide limited debugging, form painters, and other systems that enable programmers to neglect tests.

Ruby distributes REXML in its core source code, available at `http://ruby-lang.org/en/`. This project uses it to parse an XML dialect called SVG, output by GraphViz.

GraphViz in a Nutshell

"Extensible Markup Language", XML, is a portable verbose textual database format. One database it can hold is "Scalable Vector Graphics", SVG. The program GraphViz, by AT&T researchers John Ellson, Emden Gansner, Eleftherios Koutsofios, and Stephen North, contains a utility called `dag`, which means "Directed Acyclic Graphs".

We will use its successor, `dot`, which means "DAG of Tomorrow". `dot` reads descriptions of graphs, in DOT files, and lays the nodes and edges out esthetically. It produces a number of pleasant typeset outputs, such as PNG image files, Postscript files, or SVG.

GraphViz developed on Un*x, so it has RedHat packages. And it can port to MS Windows with some effort. But all we need is `dot`, and it's available (at press time) as a "Bare dot.exe". Download it from this page:

```
http://www.research.att.com/sw/tools/graphviz/download.html
TODO  or http://www.graphviz.org/Download.php
TODO  bare might be gone
```

Also get the ZIP archive labeled "third party libraries" next to it. Download and unzip that, and take out only the Dynamic Link Libraries (DLLs). Then toss `dot.exe` and all the DLLs (`ft.dll`, `jpeg.dll`, `libexpat.dll`, `libexpatw.dll`, `png.dll`, `ttf.dll`, `z.dll`, and `z.lib`) into your system folder.

To learn all `dot`'s features, read "Drawing graphs with dot", at:

```
http://www.research.att.com/sw/tools/graphviz/dotguide.pdf
```

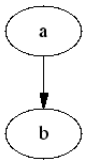
This Case Study uses only a few DOT keywords.

Architecture

At the foundation, here's a dirt-simple DOT file:

```
digraph aGraph { a -> b }
```

And here's its majestic output:



This chapter's project builds a canvas containing that output. If this were a real business application, a Logic Layer would create graphs for some reason, and a Representation Layer would convert them to DOT notation, and manage esthetic details like labels, colors, and shapes. The `dot` program would translate this notation into SVG, and our GUI Layer would display it in our new graph control.

Our mission is to create that graph control. It takes that DOT file source as an input, and displays its nodes and edges in a TkCanvas. The intermediate format will be SVG, so part of our project will be a function that renders a subset of SVG format into a TkCanvas. But we only need to recognize that subset of SVG format which `dot` can output. We will only create the prototype of a complete SVG TkCanvas; just the amount our project needs. Long before we finish enabling all possible `dot` output, the code will become very ready to extend into other SVG features.

The next Case Study adds user interactions to the canvas.

Bootstrapping

Our systems need a minimal configuration to start the test-first cycle. Create a text file called "`svgcanvas.rb`", and enter:

```
graph = 'digraph aGraph { a -> b }'
```

Ruby creates new variables by assigning them. The literal string on the right contains the DOT file notation. It says `aGraph` is a `d`irected `graph` consisting of an arrow pointing from node "`a`" to node "`b`".

We need to put that `graph` string into a file:

```
open('graph.dot', 'w') do
    |f|
    f.write(graph)
end
```

Those statements open a file called "`graph.dot`", and write the string into it. The `do` and `end` represent the Execute Around Pattern. The lines between `do` and `end` occur after `open` opens a file and before it closes it. Ruby methods, such as `open()`, can pass arguments into their `do-end` blocks using bars: `|f|` is the file "handle" object.

We might see that Execute Around Pattern again.

Save and run `svgcanvas.rb` frequently, and manually verify each new effect (until we achieve test-first).

Our next line converts the DOT notation into a file containing SVG notation:

```
system('dot -Tsvg graph.dot -o graph.svg')
```

`system()` behaves like its namesake in the C languages, Perl, etc. It shells to a command prompt and runs a command. The command invokes `dot`, telling that to translate `graph.dot` into SVG format.

The file `graph.svg` contains 23 lines of XML. The declaration for the "`a`" node alone appears like this:

```
<g id="node1" class="node"><title>a</title>
<ellipse cx="42" cy="39" rx="36" ry="24"
         style="fill:none;stroke:black;"/>
<text text-anchor="middle" x="42" y="46">a</text>
</g>
```

Only two of these <tags> create visual output: `<ellipse>` and `<text>`. The tags `<g>` and `<title>` transmit metadata. A canvas that eventually renders `dot`'s specific SVG output will be more useful than one that renders generic SVG alone. In the next Case Study, the canvas will expect that metadata, and use it to influence functions that will re-write the DOT notation.

Why we are doing all this, again? DOT notation is very high level. A programmer can express graphic intentions briefly. When our program needs to say, "The node A has a path to node B," it only needs to construct the string "`A -> B`". This is much easier than writing dozens of small manipulations and settings to render `A -> B` with the proper symmetry and spaces.

If we had a Representation Layer, `dot` would be one of its modules. Such layers typically convert a terse domain-specific representation into a verbose but generic representation. The `dot` utility provides esthetically reliable coordinates—`cx`, `cy`, etc.—for our abstract node concepts. We could easily use `dot` as a **Geometry Manager** for a dynamic GUI.

One last thing to notice in that SVG: The `<text>` tag has an attribute `text-anchor="middle"`. SVG can position text above, below, beside, or centered over an anchor point. Because `dot` will never output any other anchor, we will ignore this attribute (and more than a few like it). That laziness illustrates this important and sophisticated Agile principle:

You Aren't Gonna Need It.

Future iterations might require our SVG Canvas to interpret SVG from sources other than `dot`. We don't care. During this iteration, the user is waiting for specific features, not a full-featured generic SVG module that could render any SVG from any source. During this iteration, we will only render a narrow subset of SVG, from one source. Focusing on this iteration's features increases the odds future iterations occur.

To continue our project, we add a canvas. At first it won't do anything. One could experiment by adding `TkcText.new()` or `TkcPolygon.new()` calls here; then yank them out before proceeding:

```
require 'tk'
canvas = TkCanvas.new()
canvas.configure('width', 600)
canvas.configure('height', 600)
canvas.grid()
Tk.mainloop()
```

The `require` statement imports the Tk library.

We create a `new TkCanvas`, store a reference to it in `canvas`, and Set its size. Tk, like other GUI Toolkits, uses the Variable State Pattern to Get and Set elements of each control's long list of properties.

The line `canvas.grid()` tells Tk how to position the canvas. If the canvas has no window around it, Tk creates one automatically. The statement `Tk.mainloop()` paints the window and its canvas, and dispatches user events. Until now, the `TkCanvas` was a structure of objects in memory. `mainloop()` traverses this structure, paints each object on the screen, and blocks to await user input.

Our tests will usually address that structure of objects in memory. Between those objects and our monitor's surface are layers of code that change more slowly, for us, than our own code. Tests that focus on any of that code's behavior are less important than our tests that focus on our GUI Layer's relationship to the GUI Toolkit.

Most test cases will set up that structure of objects, query its state, and then discard it without displaying it. If we comment out our last line with a Ruby comment, `#`, and instead destroy the Canvas…

```
require 'tk'
canvas = TkCanvas.new()
canvas.configure('width', 600)
canvas.configure('height', 600)
canvas.grid()
# Tk.mainloop()
canvas.destroy()
```

…then nothing appears to happen when we run. The program exits instantly.

(Notice, per page **Error! Bookmark not defined.**, that source changes appear in **`bold`**.)

Simplest Case

To begin a hard task, solve the simplest condition within it. The simplest possible DOT graph contains nothing. Our program can already do nothing, so we step to the next-simplest condition; a single node with no label:

```
digraph aGraph { a [label = ""] }
```

That will produce a graph of a single oval. After making it work, we will add more test cases that demand other graphic primitives. We will solve those simply, then refactor their solutions together and see if the result becomes extensible into more graphics primitives.

The Ruby code to run that simplest possible graph through `dot` and produce SVG is:

```
graph = 'digraph aGraph { a [label = ""] }'
open('graph.dot', 'w') do |f|  f.write(graph)  end
system('dot -Tsvg graph.dot -o graph.svg')
```

That outputs a trivial ellipse without text. To independently verify your DOT statements, run the following command in a console to generate the file "`graph.png`", which your Web browser can show you:

```
dot -Tpng graph.dot -o graph.png
```

(Page 108 shows a test temporarily emitting such a PNG file.)

Our Ruby source code creates an SVG file:

```
system('dot -Tsvg graph.dot -o graph.svg')
```

To view it, some Web browsers come with plug-ins that display SVG format. `dot` also outputs raster formats, such as PNG. To spot-check our DOT notation, write it to a file and render it in PNG format: `dot -Tpng graph.dot -o graph.png`

Our program calls `dot` to write an SVG file, so we must read it as a string. This we pass into a function that claims to draw the SVG contents into a canvas. Then the test will check if the canvas contains … something:

```
graph = 'digraph aGraph { a [label = ""] }'
open('graph.dot', 'w') do |f|  f.write(graph)  end
system('dot -Tsvg graph.dot -o graph.svg')

svg = ''
open('graph.svg', 'r') do |f|  svg = f.read()  end

puts(svg)  #   so we can read the file

require 'tk'
canvas = TkCanvas.new()
canvas.configure('width', 600)
canvas.configure('height', 600)
canvas.grid()

putSvgIntoCanvas(svg, canvas)
all = canvas.find_all()
puts('failed')  if all.size() != 1

# Tk.mainloop()
canvas.destroy()
```

The `open()` line reads the SVG file that `dot` just wrote. The `puts()` prints the file, because for the next step we are going to need to see it, and copy elements off our console.

The function `putSvgIntoCanvas()` does not exist. Ruby is very wise and forgiving, but it draws the line at calling functions that don't exist. We will appease it:

```
def putSvgIntoCanvas(svg, canvas)
end
```

Careful inspection of that function reveals it does nothing. This is where the test-first starts. We will (generally) only write something inside that function if we can write an assertion that fails because that something is not in there. The function's choice of name and arguments represents an architectural decision.

In Programming by Intention terms, we intend a single function taking a string full of SVG notation, and a canvas. The function name uses short but complete words (and a published acronym). The argument names use the same words in the same order. After the function returns, the canvas will contain all the relevant display elements from the SVG.

Tests that demand simple interfaces will make writing the calling code, such as our `main()` function, very easy (if we ever get around to it).

We also intend test cases, but we don't have their support stuff yet. So we make due:

```
all = canvas.find_all()
puts('failed')  if all.size() != 1
```

The method `find_all()` returns a Ruby `Array` of all the elements in the canvas, as object references.

The line `puts('failed')  if all.size() != 1` is our primordial assertion. Soon we will upgrade it into a true `assert()` statement. Ruby, like Perl, uses "statement modifiers" to control flow on a single line, with the `if` after the controlled statement.

To pass that lowly test, we need to learn to create Ovals in Tk-land:

```
def putSvgIntoCanvas(svg, canvas)
    TkcOval.new(canvas, 10, 10, 20, 20)
end
```

That function is still incomplete. It does not read the SVG commands inside the string `svg` to produce its oval. But it makes the tests pass. So are the tests fooling us? Or is the code?

As a recap, here is the complete prototypical application and its single assertion:

```
require 'tk'

def putSvgIntoCanvas(svg, canvas)
    TkcOval.new(canvas, 10, 10, 20, 20)
end

graph = 'digraph aGraph { a [label = ""] }'
open('graph.dot', 'w') do |f|  f.write(graph)  end
system('dot -Tsvg graph.dot -o graph.svg')

svg = ''
open('graph.svg', 'r') do |f|  svg = f.read()  end

puts(svg)  #   so we can read the file

canvas = TkCanvas.new()
canvas.configure('width', 600)
canvas.configure('height', 600)
canvas.grid()

putSvgIntoCanvas(svg, canvas)
all = canvas.find_all()
puts('failed')  if all.size() != 1  # <-- assertion
# Tk.mainloop()

canvas.destroy()
```

We can de-comment `Tk.mainloop()` at the bottom to see the tiny oval. But it's not in the same location as the oval `dot` wrote into the SVG code.

The Test Rig

The rules say we may not add a feature to the code until the tests demand that feature. When we wrote `TkcOval.new()` we stretched those rules—our tests do not even require an oval yet. Giving the new oval bogus coordinates stretches no rules. Lying is part of the TFP cycle.

Before making the test more aggressive, we must replace `puts('failed')` with a real `assert` statement:

```
assert_equal(all.size(), 1)
```

Because we expect to have quite a few more assertions, this upgrade is proactive duplication removal.

Ruby's leading test rig is Ruby::Unit, by Nathaniel Talbott. After we import it and wrap it around our methods, the new lines are **bold**:

```
require 'tk'
require 'test/unit'

def putSvgIntoCanvas(svg, canvas)
    TkcOval.new(canvas, 10, 10, 20, 20)
end

class  TestGrapher < Test::Unit::TestCase
    def test_oneNode
        graph = 'digraph aGraph { a [label = ""] }'
        open('graph.dot', 'w') do |f|  f.write(graph)  end
        system('dot -Tsvg graph.dot -o graph.svg')

        svg = ''
        open('graph.svg', 'r') do |f|  svg = f.read()  end

        puts(svg)  #   so we can read the file

        canvas = TkCanvas.new()
        canvas.configure('width', 600)
        canvas.configure('height', 600)
        canvas.grid()

        putSvgIntoCanvas(svg, canvas)
        all = canvas.find_all()

        assert_equal(all.size(), 1,
            'the canvas should have at least something')

 #       Tk.mainloop()

        canvas.destroy()
    end
end
```

Purists will note the testee still lives in the same file as the test. Ruby lets us keep it here (the **Self Test pattern**), or we may move it out at any time. Other languages, or their test rigs, may insist their test targets live in modules separate from their test suites. For our project, at this early stage, we will leave the testee in the same file, and move it to another file when that becomes more convenient. When tests make code easy to refactor, both logical and physical design changes are safe.

The new `require` lines pull in the `Test::Unit::TestCase` class. Our class, `TestGrapher`, inherits that class via `< Test::Unit::TestCase`. The little `<` mark means, "Push all of that class's abilities into my class."

`Test::Unit` uses the Test Collector pattern to seek every class that inherits `Test::Unit`, and automatically run their every method beginning with "`test`".

Per the common overview of unit test frameworks, on page 452, our test rig performs these actions for each `test_` method:

- Construct a new `TestGrapher` object
- Call `aTestGrapher.setup()`
- Call `aTestGrapher.test_`*`whatever`*`()`
- Call `aTestGrapher.teardown()`

`TestCase`, the base class, provides empty implementations of `setup()` and `teardown()`. We did not override them yet, so for our first test they do nothing.

Force the Code to Do More

We feel bad about the contents of `putSvgIntoCanvas()`. That function cheats; it does not use the numbers inside the `svg` string. To feel closure (and eliminate behavioral duplication between the two groups of coordinates), we'll add an XML parser and use it to fetch the numbers. We use REXML 2 by Sean Russell, and its XPath module. It comes bundled with Ruby; its home page is:

```
http://www.germane-software.com/software/XML/rexml_stable/
```

As with `TkCanvas` and `dot`, before I throw REXML into this project I first , to teach myself how REXML works. I spend time playing with REXML's sample programs, and writing experimental statements & Learner Tests. These are not shown.

After erasing these experiments, we return to the test. It needs assertions that use REXML to compare the size of SVG's ellipse to the size of the TkCanvas's `TkcOval`.

Do the paperwork:

```
require 'rexml/document'
include REXML
```

Next, we add test statements to extract the first ellipse, gets its center and radii, and converts them into a bounding box, which we `expect` the test to match:

```
        putSvgIntoCanvas(svg, canvas)
        all = canvas.find_all()
        assert_equal(all.size(), 1)

        oval = all[0]
        doc = Document.new(svg)

        ellipse = XPath.first(doc, '/svg/g/g/ellipse')
        cx = ellipse.attributes['cx'].to_i()
        cy = ellipse.attributes['cy'].to_i()
        rx = ellipse.attributes['rx'].to_i()
        ry = ellipse.attributes['ry'].to_i()
        expect = [cx - rx, cy - ry, cx + rx, cy + ry]

        assert_equal(expect, oval.coords(),
            'the oval should be where the SVG specified')
```

```
        # Tk.mainloop()
```

Hurray! It fails! We must soon add some code, and make the test pass. First I explain the new lines.

XPath in a Nutshell

`oval` is a `TkCanvas` element. `ellipse` is a REXML object; the first ellipse found inside the SVG tags. The XPath query "`/svg/g/g/ellipse`" found it by walking the XML tag structure:

```
<svg>...<g>...<g>...<ellipse...>...</ellipse>...</g>...</g>...</svg>
```

`XPath.first()` returns `ellipse`, an object populated with all the shaded data in that sample XML. The `ellipse` object stores, as member data, the tag name, attributes, and contents. XML parsers strip out the <> marks, so XPath doesn't see them.

The actual data, in the XML source, are "`<ellipse cx="42" cy="39" rx="36" ry="24" style="fill:none;stroke:black;"/>`".

These next four lines pull out the ellipse's center location and two radii, and turn them into_integers:

```
        cx = ellipse.attributes['cx'].to_i()
        cy = ellipse.attributes['cy'].to_i()
        rx = ellipse.attributes['rx'].to_i()
        ry = ellipse.attributes['ry'].to_i()
```

Unlike SVG items, `TkCanvas` items position by the inverted Cartesian coordinates of their upper left and lower right corner. Tk expects these coordinates unrolled into a flat array, as `[x1,y1,x2,y2]`. That's what `oval.coords()` returns.

Ruby::Unit Fault Navigation

When the assertion fails our expectation, it prints this diagnostic:

```
Failure!!!
ForceTheCodeToDoMore.rb:41:test_oneNode(TestGrapher):
the oval should be where the SVG specified.
<[6, 15, 78, 63]> expected but was
<[10.0, 10.0, 20.0, 20.0]>
```

If an assertion succeeds it prints nothing. If it fails, it prints the name of the test case, the name of the test suite, and the file name & line number.

The assertion then prints its commentary string: "`the oval should be where the SVG specified.`" This feature helps engineers rapidly assess failures. Engineers should give their assertions more commentary strings than I will in this book:

```
assert_equal(expect, oval.coords(),
    'the oval should be where the SVG specified')
```

I won't use that feature again. Compilers worry about right margins less than books do. Blocks of assertions should be easy to scan, but if I must keep inserting line feeds and commentary strings, then the blocks of assertions in this book will be hard to scan. Please do as I say, not as I do!

TODO TDD thread what's the commentary string called?

A commentary string helps programmers understand the failure. Because failure is negative and assertions are positive etc. TODO

The assertion next `inspect`s each of its arguments, with verbiage between them based on the kind of assertion involved. `assert_equal()` prints "`expected but was`" in our situation.

The best form of assertion would also print the source code of the expression that failed:

```
Failure!!!
test_oneNode(TestGrapher) [sample.rb:41]:
the oval should be where the SVG specified.
expect was <[6, 15, 78, 66]> but
oval.coords() was <[10.0, 10.0, 20.0, 20.0]>
```

This style reduces the need for a commentary string; your variable names should self-document anyway. Ruby::Unit does not have this feature, but there should be ways to add it. Use the Source, Luke!

Get Back to a Green Bar

As I build this Case Study, I don't use a graphical test runner (such as `Test::Unit::UI::Tk::TestRunner`). I don't have a real Green Bar. This early in a project, you might not even have an editor that launches a Ruby program from one button. Future Case Studies will cover *One Button Testing*. If you play along with this Case Study, you might have an editor enabled for Ruby (such as `scite`, which comes with the Pragmatic Programmer's installer). Or you might use a simple text editor. Saving a file, switching to a console, and running a command line manually requires more than one test button. We introduce a few Principles at a time, so pretend that we have a real Test Button, and real Green Bars and Red Bars.

Our projects' test would present a Red Bar, because our oval is not in the right location. The feeling of Red Bar urgency inspires us to fix that code and get a Green Bar as soon as possible, by any means necessary.

To make the test pass, copy some code from the test into that bogus function:

```
def putSvgIntoCanvas(svg, canvas)
    doc = Document.new(svg)
    ellipse = XPath.first(doc, '/svg/g/g/ellipse')
    cx = ellipse.attributes['cx'].to_i()
    cy = ellipse.attributes['cy'].to_i()
    rx = ellipse.attributes['rx'].to_i()
    ry = ellipse.attributes['ry'].to_i()
    coordinates = [cx - rx, cy - ry, cx + rx, cy + ry]
    TkcOval.new(canvas, coordinates)
```

```
end
```

The tests pass, and the function is less bogus.

Copying the test code into the class is not cheating; it may ultimately be pragmatic. But both copies might also now bear the same, matching error. See the Principle *Acolyte Checks Output*, on page 35.

TODO coords, coordinates, etc.?

When we have passing tests, they can support refactors. Check for duplication, starting with the production code.

The statements that all say "**`?? = ellipse.attributes['??'].to_i()`**" look the same. The fix uses Ruby idiomatically. `.collect()` converts one array into another:

```
def collectCoordinates(tkc, coordinateNames)
    return coordinateNames.collect do |name|
        tkc.attributes[name].to_i() end
end

def putSvgIntoCanvas(svg, canvas)
    doc = Document.new(svg)
    ellipse = XPath.first(doc, '/svg/g/g/ellipse')

    cx,cy,rx,ry = collectCoordinates(ellipse,
                          ['cx','cy','rx','ry'])

    coordinates = [cx - rx, cy - ry, cx + rx, cy + ry]
    TkcOval.new(canvas, coordinates)
end
```

This new version of `putSvgIntoCanvas()` does the same thing as the previous version did. Our tests concur.

Close the Loop

The Bootstrapping is over. We have source that accepts input (DOT notation), and displays output (an oval in a TkCanvas). The stations around this loop have test cases, and the cases have a test rig. To add each remaining feature, we visit each station in this loop and upgrade it.

Feature Accretion

In math theory, graphs have nodes, edges, and sometimes attributes. In GraphViz, graph nodes have text attributes. We implement those next:

```
 def test_oneNode**WithText**
    graph = 'digraph aGraph { a [label="**text**"] }'
    open('graph.dot', 'w') do |f| f.write(graph) end
    system 'dot -Tsvg graph.dot -o graph.svg'
…
   # Tk.mainloop()
    canvas.destroy()
end
```

Of course it's just the prior test, cloned, with edits in **bold**. The big difference is our graph's single node has a text label inside it.

This test passes. In general, writing new tests is less suspicious than cloning them. Our first clone passed its tests because the assertions matched the new bugs. We clone and modify tests to increase the odds we can create test fixtures from them. Test code style differs subtly from production code.

The cloned test worked because its modifications only requested new features of `dot`, not our own code. But we should run it and see it pass anyway, to ensure the new commands call `dot` correctly.

> The more edits you make, the more anxious you become, wondering whether you can accurately predict the next test result.

After the half-done test passes, make it request the text feature:

```
            all = canvas.find_all()
            assert_equal(all.size(), 2)
...
            cnvText = all[1]
            text = XPath.first(doc, '/svg/g/g/text')
            x = text.attributes['x'].to_i()
            y = text.attributes['y'].to_i()
            expect = [x, y]
            assert_equal(expect, cnvText.coords())
            assert_equal('text', cnvText.cget('text'))
```

(The entire program listing appears soon.)

That code uses an XPath query, "`/svg/g/g/text`", to extract a `<text>` tag out of the SVG. The assertions demand that the second item on the canvas is a `TkcText` object with the correct coordinates, and a `text` configuration containing "`text`". (That sample data value, in retrospect, could have been better...)

Now here's the code that makes the test pass:

```
def putSvgIntoCanvas(svg, canvas)
    doc = Document.new(svg)
    ellipse = XPath.first(doc, '/svg/g/g/ellipse')

    cx,cy,rx,ry = collectCoordinates(ellipse,
                          ['cx','cy','rx','ry'])

    coordinates = [cx - rx, cy - ry, cx + rx, cy + ry]
    TkcOval.new(canvas, coordinates)

    svgText = XPath.first(doc, '/svg/g/g/text')

    if svgText != nil then
        x = svgText.attributes['x'].to_i()
        y = svgText.attributes['y'].to_i()
        coordinates = [x, y]

        TkcText.new(canvas, x, y) do
                text(svgText.text())
                anchor('center')
            end
    end
end
```

All tests pass, and when you de-comment the `Tk.mainloop()` call, you see an oval with text inside it.

An early push to pass new kinds of tests often fills code up with many flaws, both technical and esthetic. If a DOT script generated more than one oval, for example, the call to `XPath.first()` would only draw the first one.

Refactor Low Hanging Fruit

> These are all ripe situations for new tests to force the code's technique to improve. However, the TFP principles say we must first improve the style before adding more tests.

The current style contains much room for improvement.

TODO We practice these simple rules on small projects, even if we could easily guess the resulting design. Incremental upgrades might be the only hope for applications too large to fit in our short-term memory. These simple rules help applications not grow large, and not appear large when they cover many features.

A design flaw in a more complicated situation might be less obvious. Of course we know that `XPath.first()` will soon convert to `XPath.each()` (where `.each()` is Ruby's iteration method, like `for each` in other languages). So we will pretend we forgot that, then see how following the rules corrects the code.

If the code did not have a book, it should have more comments. Really.

Here is the result of all the edits so far:

```
require 'tk'
require 'test/unit'
require 'rexml/document'
include REXML

def collectCoordinates(tkc, coordinateNames)
    return coordinateNames.collect do |name|
        tkc.attributes[name].to_i() end
end

def putSvgIntoCanvas(svg, canvas)
    doc = Document.new(svg)
    ellipse = XPath.first(doc, '/svg/g/g/ellipse')

    cx,cy,rx,ry = collectCoordinates(ellipse,
                          ['cx','cy','rx','ry'])

    coordinates = [cx - rx, cy - ry, cx + rx, cy + ry]
    TkcOval.new(canvas, coordinates)

    svgText = XPath.first(doc, '/svg/g/g/text')
```

```
    if svgText != nil then
        x = svgText.attributes['x'].to_i()
        y = svgText.attributes['y'].to_i()
        coordinates = [x, y]

        TkcText.new(canvas, x, y) do
                text(svgText.text())
                anchor('center')
            end
    end
end

class  TestGrapher < Test::Unit::TestCase

    def test_oneNodeWithText
        graph = 'digraph aGraph { a [label="text"] }'
        open('graph.dot', 'w') do |f| f.write(graph) end
        system 'dot -Tsvg graph.dot -o graph.svg'

        svg = ''
        open('graph.svg', 'r') do |f|  svg = f.read()  end


    #    puts(svg)  #   so we can read the file

        canvas = TkCanvas.new()
        canvas.configure('width', 600)
        canvas.configure('height', 600)
        canvas.grid()

        putSvgIntoCanvas(svg, canvas)
        all = canvas.find_all()
        assert_equal(all.size(), 2)

        oval = all[0]
        doc = Document.new(svg)

        ellipse = XPath.first(doc, '/svg/g/g/ellipse')
        cx = ellipse.attributes['cx'].to_i()
        cy = ellipse.attributes['cy'].to_i()
        rx = ellipse.attributes['rx'].to_i()
        ry = ellipse.attributes['ry'].to_i()
        expect = [cx - rx, cy - ry, cx + rx, cy + ry]
        assert_equal(expect, oval.coords())

        cnvText = all[1]
        text = XPath.first(doc, '/svg/g/g/text')
        x = text.attributes['x'].to_i()
        y = text.attributes['y'].to_i()
        expect = [x, y]
        assert_equal(expect, cnvText.coords())
        assert_equal('text', cnvText.cget('text'))

        # Tk.mainloop()
        canvas.destroy()
    end
```

```
    def test_oneNode
        graph = 'digraph aGraph { a [label = ""] }'
        open('graph.dot', 'w') do |f|  f.write(graph)  end
        system('dot -Tsvg graph.dot -o graph.svg')

        svg = ''
        open('graph.svg', 'r') do |f|  svg = f.read()  end

        canvas = TkCanvas.new()
        canvas.configure('width', 600)
        canvas.configure('height', 600)
        canvas.grid()

        putSvgIntoCanvas(svg, canvas)
        all = canvas.find_all()
        assert_equal(all.size(), 1)

        oval = all[0]
        doc = Document.new(svg)

        ellipse = XPath.first(doc, '/svg/g/g/ellipse')
        cx = ellipse.attributes['cx'].to_i()
        cy = ellipse.attributes['cy'].to_i()
        rx = ellipse.attributes['rx'].to_i()
        ry = ellipse.attributes['ry'].to_i()
        expect = [cx - rx, cy - ry, cx + rx, cy + ry]
        assert_equal(expect, oval.coords())

#       Tk.mainloop()

        canvas.destroy()
    end
end
```

Those two test functions look very similar. Squinting at the lines without reading them reveals they duplicate the same general shapes. Practice many ways to notice duplication, including observing the general shapes of functions.

The tests follow the Assemble Act Assert pattern. Each test Assembles target objects (in many lines at the top of each test); Activates their function (`putSvgIntoCanvas()`), and then Asserts the function had good results and side effects. Our new tests duplicate Assembly.

To fix the design, find the smallest things that duplicate and pull them out to make little methods. Proceed in the smallest testable increments possible. For example, if we observe that both tests create a canvas, we could pull the duplicated creation out into a common method:

```
def createCanvas()
    canvas = TkCanvas.new()
    canvas.configure('width', 600)
    canvas.configure('height', 600)
    canvas.grid()
    return canvas
end
```

We write that new method, then test. Following the Extract Method Refactor, we leave the other places in the code alone; we don't replace the other calls to `TkCanvas.new()` and the lines around them with a call to `createCanvas()` yet. We test first, to ensure that the new method compiles correctly, and does nothing to derail the tests.

Now find one place that calls `TkCanvas.new()`, and replace all the lines around it with a call to `createCanvas()`:

```
svg = ''
open('graph.svg', 'r') do |f|  svg = f.read()  end
canvas = createCanvas()
putSvgIntoCanvas(svg, canvas)
```

Make sure the tests still pass, find the next place, and repeat. Make as many opportunities as possible to test.

To help this narrative, going forward, we won't mention each intermediate refactoring step. This Case Study's online source reveals most steps.

Our test cases duplicate the lines that generate SVG. Put them into a function:

```
def generateSvg(graph)
    open('graph.dot', 'w') do |f| f.write(graph) end
    system 'dot -Tsvg graph.dot -o graph.svg'

    open('graph.svg'2, 'r') do |f|
        return f.read()  end

end
```

Regulate the Tk Event Queue

Now we come to the end of both test cases, and meet a peculiar formation. It appears twice, so the *Merge Duplicated Behavior* Simplicity Principle requires the Extract Method Refactor to merge it into a common method. Then both cases will share it:

```
   # Tk.mainloop()
    canvas.destroy()
```

It has a commented line in it. Programmers often use their tests as a "programmer's user interface" into the code. Programmers switch lines on and off with comment markers, the same as users click checkboxes.

Our switch must raise to a level higher than a comment. Consider this potential shared method:

```
def maybeMainloop(canvas)
   # Tk.mainloop()
    canvas.destroy()
end
```

If many cases used that fixture, and if we took out the `#` comment mark to see a canvas, every case would display its canvas. Tk would put them all the same window; that would resemble a bug. When you view a window to check its contents, its appearance must not mislead.

Make the switch more explicit:

```
def maybeMainloop(canvas, reveal = false)
    if reveal then
        Tk.mainloop()
    else
        canvas.destroy()
    end
end
```

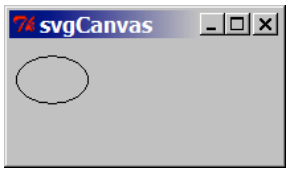
(Test.) Change the end of one test to this:

```
    maybeMainloop(canvas)
```

(Test.) And change the end of another test to this:

```
    maybeMainloop(canvas, true)
```

Run the tests, and we see only one of the canvases pop up. It still has an oval in it:



`maybeMainloop()` *Regulates* our *Event Queue*, supporting *Temporary Visual Inspections* and *Interactive Tests*. It doesn't follow all the Principles yet; closing the window might crash the tests. It works well enough for our simple project, using an uncommon library. Other Case Studies will develop similar system called `revealFor()` that follows all those rules.

The next few refactors, not shown, make `canvas` a member variable of the test case. (Ruby prefixes member variables with `@`.) That simplifies the arguments to `maybeMainloop()`; it's another example of folding duplication together. That permits `createCanvas()` to rename to `setup()`. Now `Test::Unit::TestCase` creates a new canvas before every test case, and they all have fewer lines.

The online source contains all these refactors. They consist of >20 individual test runs. Most don't display that window. Displaying user interfaces when the need and ability maximize permit long periods of test-first development without displaying user interfaces.

Here's the complete program. I added one new (failing) test on the end:

```
require 'tk'
require 'test/unit'
require 'rexml/document'
include REXML

def collectCoordinates(tkc, coordinateNames)
    return coordinateNames.collect do |name|
        tkc.attributes[name].to_i() end
end
```

```
def putSvgIntoCanvas(svg, canvas)
    doc = Document.new(svg)
    ellipse = XPath.first(doc, '/svg/g/g/ellipse')

    cx,cy,rx,ry = collectCoordinates(ellipse,
                          ['cx','cy','rx','ry'])

    coordinates = [cx - rx, cy - ry, cx + rx, cy + ry]
    TkcOval.new(canvas, coordinates)

    svgText = XPath.first(doc, '/svg/g/g/text')

    if svgText != nil then
        x = svgText.attributes['x'].to_i()
        y = svgText.attributes['y'].to_i()
        coordinates = [x, y]

        TkcText.new(canvas, x, y) do
                text(svgText.text())
                anchor('center')
            end
    end
end

def generateSvg(graph)
    open('graph.dot', 'w') do |f| f.write(graph) end
    system 'dot -Tsvg graph.dot -o graph.svg'

    open('graph.svg', 'r') do |f|
        return f.read()  end
end

class  TestGrapher < Test::Unit::TestCase

    def setup()
        @canvas = TkCanvas.new()
        @canvas.configure('width', 600)
        @canvas.configure('height', 600)
        @canvas.grid()
        return @canvas  #  TODO  yank this out
    end

    def maybeMainloop(reveal = false)
        if reveal then
            Tk.mainloop()
        else
            @canvas.destroy()
        end
    end

    def test_oneNodeWithText
        graph = 'digraph aGraph { a [label="text"] }'
        svg = generateSvg(graph)

        putSvgIntoCanvas(svg, @canvas)
        all = @canvas.find_all()
        assert_equal(all.size(), 2)

        oval = all[0]
        doc = Document.new(svg)
```

```
        ellipse = XPath.first(doc, '/svg/g/g/ellipse')
        cx = ellipse.attributes['cx'].to_i()
        cy = ellipse.attributes['cy'].to_i()
        rx = ellipse.attributes['rx'].to_i()
        ry = ellipse.attributes['ry'].to_i()
        expect = [cx - rx, cy - ry, cx + rx, cy + ry]
        assert_equal(expect, oval.coords())

        cnvText = all[1]
        text = XPath.first(doc, '/svg/g/g/text')
        x = text.attributes['x'].to_i()
        y = text.attributes['y'].to_i()
        expect = [x, y]
        assert_equal(expect, cnvText.coords())
        assert_equal('text', cnvText.cget('text'))

        maybeMainloop()
    end

    def test_oneNode
        graph = 'digraph aGraph { a [label = ""] }'
        svg = generateSvg(graph)

        putSvgIntoCanvas(svg, @canvas)
        all = @canvas.find_all()
        assert_equal(all.size(), 1)

        oval = all[0]
        doc = Document.new(svg)

        ellipse = XPath.first(doc, '/svg/g/g/ellipse')
        cx = ellipse.attributes['cx'].to_i()
        cy = ellipse.attributes['cy'].to_i()
        rx = ellipse.attributes['rx'].to_i()
        ry = ellipse.attributes['ry'].to_i()
        expect = [cx - rx, cy - ry, cx + rx, cy + ry]
        assert_equal(expect, oval.coords())

        maybeMainloop()
    end

    def test_twoNodes()
        svg = generateSvg(
            'digraph aGraph { a [label=""]; b [label=""]; }')

        putSvgIntoCanvas(svg, @canvas)
        all = @canvas.find_all()
        assert_equal(all.size(), 2)
        assert_equal(all[0].class.name(), 'TkcOval')
        assert_equal(all[1].class.name(), 'TkcOval')
        maybeMainloop()
    end

end
```

Test-away a Bug

Previously we noticed that the function `putSvgIntoCanvas()` would fail if we created more than one node. Our new test expresses this problem. It creates a graph with two nodes (and no text inside them). Then our test asserts that we have `2` items in the canvas, and both of their class types are `TkcOval`. (Note that a real project's requirements might eventually lead to the

situation “a graph with two nodes”. If we did not notice the problem with the code, other development paths would converge on the same fix.)

The newly-added test, on the previous page, fails. The fix converts `first()` to `each()`. In Ruby, a method that calls its block (between `do` and `end`) many times is typically called “`each`”:

```
    def putSvgIntoCanvas(svg, canvas)
        doc = Document.new(svg)

        XPath.each(doc, '/svg/g/g/ellipse') do
            |ellipse|

            cx,cy,rx,ry = collectCoordinates(ellipse,
                      ['cx','cy','rx','ry'])

            coordinates = [cx - rx, cy - ry, cx + rx, cy + ry]
            TkcOval.new(canvas, coordinates)
        end

        svgText = XPath.first(doc, '/svg/g/g/text')

        if svgText != nil then
            x = svgText.attributes['x'].to_i()
            y = svgText.attributes['y'].to_i()
            coordinates = [x, y]

            TkcText.new(canvas, x, y) do
                    text(svgText.text())
                    anchor('center')
                end
        end
    end
```

Another test, called `test_twoNodesWithText()`, will force us to edit the statement `XPath.first(doc, '/svg/g/g/text')` into `XPath.each(doc, '/svg/g/g/text')`.

Here’s the test:

```
 def test_twoNodesWithText()
        svg = generateSvg(
            'digraph aGraph { a [label="Harry Potter"]; ' +
                'b [label="and the Spiders from Mars"]; }')

        putSvgIntoCanvas(svg, @canvas)
        all = @canvas.find_all()
        assert_equal(all.size(), 4)
        assert_equal(all[0].class.name(), 'TkcOval')
        assert_equal(all[1].class.name(), 'TkcOval')
        assert_equal(all[2].class.name(), 'TkcText')
        assert_equal(all[3].class.name(), 'TkcText')
        maybeMainloop(true)
    end
```

Observe two things: The labels are different lengths, and `maybeMainloop(true)` will reveal the canvas for our approval. We don't yet have a test that checks the second oval or text use their correct coordinates. If we viewed them, and they displayed a bug, our index of suspicion would rise, and we would test more.

Similarly, suppose we developed our Representation Layer concurrently, instead of using dot, which is published. Then its esthetics might be under development, and they would also raise our index of suspicion.

Our text labels are different lengths. If they had caught a bug—if, for example, the second oval borrowed the first one's dimensions—then the text might overflow or otherwise appear obviously wrong. Turn on `maybeMainloop(true)`, and see:

Whimsical sample data keep us awake, ensure test runs do not resemble production runs, and prevent us from thinking too hard about inconsequential things. Don't sit around pondering what to write; always just write the same dumb sample data for various tests. My colleagues have learned to trace stray instances of the words "kozmik" and "bullfrog" to me.

Passing those tests produced this code fragment, with some other improvements slipped in:

```
    XPath.each(doc, '/svg/g/g/text') do
        |svgText|

        x, y = collectCoordinates(svgText, ['x','y'])
        tx = TkcText.new(canvas, x, y)
        tx.configure('text', svgText.text())
        tx.configure('anchor', 'center')
    end
```

TODO dig up and insert the old "cosmic bullfrog" comic here (not by me).

Framework

We have neglected designing our most important method, `putSvgIntoCanvas()`. By selecting the right duplications within it to fold together, we can force it to use polymorphism. This can make adding new features easier. Here's that function's current version:

```
def putSvgIntoCanvas(svg, canvas)
    doc = Document.new(svg)

    XPath.each(doc, '/svg/g/g/ellipse') do
        |ellipse|

        cx,cy,rx,ry = collectCoordinates(ellipse,
                  ['cx','cy','rx','ry'])

        coordinates = [cx - rx, cy - ry, cx + rx, cy + ry]
        TkcOval.new(canvas, coordinates)
    end

    XPath.each(doc, '/svg/g/g/text') do
        |svgText|

        coordinates = collectCoordinates(svgText, ['x','y'])
        tx = TkcText.new(canvas, coordinates)
        tx.configure('text', svgText.text())
        tx.configure('anchor', 'center')
    end
end
```

This function is very long—it already exceeds Common Style Guides. And if we foresee other graphic primitives, such as the curves and polygons that `dot`'s SVG output might uses, the function would grow much longer.

TODO add the "replace conditionals with polymorphism" lecture

The function duplicates code in ways that suggest polymorphism might improve it. The code shows blocks with a similar shape, but some lines inside each block are different. This indicates a base class method ought to house the common code from the blocks, and call virtual methods in derived classes containing the specific behaviors.

But we first obey the "lowest hanging fruit" principle, and only refactor what's obvious and easy. If we foresee an advanced design, we can only reach for it after it becomes the easiest refactor available. Clearing those easy refactors out of the way makes the harder ones easy too.

The lines that generate each `coordinates` variable would be simple to refactor. They would need one argument, and return one result. We copy them out into smaller methods, but we only write the new methods, and leave the original coordinate statements alone:

```
def getEllipseCoordinates(ellipse)
    cx,cy,rx,ry = collectCoordinates(ellipse,
              ['cx','cy','rx','ry'])

    return [cx - rx, cy - ry, cx + rx, cy + ry]
end

def getTextCoordinates(text)
    return collectCoordinates(text, ['x','y'])
end
```

Inspecting those two new method's names, we notice that they duplicate the words "`get`" and "`Coordinates`". Keep that in mind. Duplication inside (well-named) identifiers is still duplication.

After the tests pass, now replace the original coordinate statements with these methods.

The function `putSvgIntoCanvas()` is now shorter, but it still duplicates much mechanics around the calls to `XPath.each()`:

```
def putSvgIntoCanvas(svg, canvas)
    doc = Document.new(svg)

    XPath.each(doc, '/svg/g/g/ellipse') do
        |ellipse|

        coordinates = getEllipseCoordinates(ellipse)
        TkcOval.new(canvas, coordinates)
    end

    XPath.each(doc, '/svg/g/g/text') do
        |svgText|

        coordinates = getTextCoordinates(svgText)
        tx = TkcText.new(canvas, coordinates)
        tx.configure('text', svgText.text())
        tx.configure('anchor', 'center')
    end
end
```

The two big parts of this function are very similar, but not the same. Our next refactor makes them look *more* similar. This gets them ready for duplication removal. So, after several refactors and tests:

```
def newEllipse(canvas, svgEllipse)
    coordinates = getEllipseCoordinates(svgEllipse)
    TkcOval.new(canvas, coordinates)
end

def newText(canvas, svgText)
    coordinates = getTextCoordinates(svgText)
    tx = TkcText.new(canvas, coordinates)
    tx.configure('text', svgText.text())
    tx.configure('anchor', 'center')
end

def putSvgIntoCanvas(svg, canvas)
    doc = Document.new(svg)

    XPath.each(doc, '/svg/g/g/ellipse') do
        |svgEllipse|
        newEllipse(canvas, svgEllipse)
    end

    XPath.each(doc, '/svg/g/g/text') do
        |svgText|
        newText(canvas, svgText)
     end
end
```

We have delayed making a relatively obvious class out of these common behaviors for long enough. Our functions now have similar names and parallel internals; eventually they might fit the Abstract Template Pattern. That means a concrete method contains an algorithm (such as a loop statement or a call to a constructor), and some statements inside this method are calls to virtual methods that derived classes override. The benefit is the algorithm appears once, but derived classes customize its exact behavior. (And rest assured I forced refactoring to make this pattern appear, by merging duplicated behavior, before coming back up here and describing it!)

First, write classes whose only job is to return the names of their SVG shape types, and then move `newEllipse()` and `newText()` into them. Install these classes into their calling sites, but don't call them polymorphically yet. A good name for misused classes like these is "**Compilable Comments**", but we only temporarily misuse them:

```
class EllipseBuilder
    def svgTag();  return 'ellipse';  end

    def newEllipse(canvas, svgEllipse)
        coordinates = getEllipseCoordinates(svgEllipse)
        TkcOval.new(canvas, coordinates)
    end
end

class TextBuilder
    def svgTag();  return 'text';  end

    def newText(canvas, svgText)
        coordinates = getTextCoordinates(svgText)
        tx = TkcText.new(canvas, coordinates)
        tx.configure('text', svgText.text())
```

```
        tx.configure('anchor', 'center')
    end
end

def putSvgIntoCanvas(svg, canvas)
    doc              = Document.new(svg)
    anEllipseBuilder = EllipseBuilder.new()
    aTextBuilder     = TextBuilder.new()

    XPath.each(doc, '/svg/g/g/' + anEllipseBuilder.svgTag()) do
        |svgEllipse|
        anEllipseBuilder.newEllipse(canvas, svgEllipse)
    end

    XPath.each(doc, '/svg/g/g/' + aTextBuilder.svgTag()) do
        |svgText|
        aTextBuilder.newText(canvas, svgText)
    end
end
```

TODO puts something about "variables should have the narrowest scope possible here.

(As a style note, some methods only return a literal value, such as 'text'. I pack them on one line to get them out of the way:

```
    def svgTag();  return 'text';  end
```

If that method contained logic, it would require multiple lines for clarity.)

The last refactor hardly improved things. All it did was move elements that duplicate into classes that duplicate.

The design is about to take a quantum leap for the better. I will illustrate each small motion, because intermediate refactors must leave a design halfway between two patterns, with passing tests, and how to do this is not always intuitive. The design briefly gets much worse.

First, we build an array and put only one of the "builders" in it:

```
    builders = [anEllipseBuilder]
```

Test. Then traverse the array, and do nothing:

```
        builders.each() do |aBuilder|
            end
```

Test. Now we need to stop calling `anEllipseBuilder`'s methods directly, and start calling the same methods through `aBuilder`. So take one of the blocks, put it inside, change some names, and test again:

```
def putSvgIntoCanvas(svg, canvas)
    doc              = Document.new(svg)
    anEllipseBuilder = EllipseBuilder.new()
    aTextBuilder     = TextBuilder.new()

    builders = [anEllipseBuilder]

    builders.each() do |aBuilder|
        XPath.each(doc, '/svg/g/g/' + aBuilder.svgTag()) do
            |svgThing|
```

```
            aBuilder.newThing(canvas, svgThing)
        end
    end

    XPath.each(doc, '/svg/g/g/' + aTextBuilder.svgTag()) do
        |svgText|
        aTextBuilder.newThing(canvas, svgText)
    end
end
```

At this point, our design is half one pattern and half another. The tests don't care; they still pass. If our boss interrupts with a request to demo or deliver, right now, the project is ready for integration, and for quality control to inspect it. If it had enough features, it would be ready for users to start using it, right now. None of them care the design is half one pattern and half another.

Large, crufty applications that delay refactoring may require extensive refactors to improve. Chronic testing enables long refactors without giving our colleagues or customers long blackouts. However, this Case Study refactored as soon as its design started to grow crufty, not long after. Clean a kitchen after each meal (instead of after a food poisoning episode) to make each meal preparation time the same. Chronic refactoring prevents the need for extensive refactors.

> The more often you refactor toward Object Oriented design guidelines, the less often you will need more refactoring. A stitch in time saves nine.

Here's where we finally introduce the better design. We just do the same refactor to `aTextBuilder`:

```
def putSvgIntoCanvas(svg, canvas)
    doc              = Document.new(svg)
    anEllipseBuilder = EllipseBuilder.new()
    aTextBuilder     = TextBuilder.new()

    builders = [anEllipseBuilder, aTextBuilder]

    builders.each() do |aBuilder|
        XPath.each(doc, '/svg/g/g/' + aBuilder.svgTag()) do
            |svgThing|
            aBuilder.newThing(canvas, svgThing)
        end
    end

 #   XPath.each(doc, '/svg/g/g/' + aTextBuilder.svgTag()) do
 #           |svgText|
 #           aTextBuilder.newThing(canvas, svgText)
 #       end
end
```

Sometimes I comment code out first, test, then delete it. I trust my editor's Undo button. Really. But if the test had failed due to a missing element from the commented-out code, I'd like the evidence to appear readily on the screen. I don't want to accidentally hit Undo too many times and restore too much. If I had an editor that bonded its Undo mechanism with "snapshots" of the code state for each passing test, I would comment code out less often.

Now we turn our attention to the builder classes. They still look like this:

```
class EllipseBuilder
    def svgTag();  return 'ellipse';  end

    def newThing(canvas, svgEllipse)
        coordinates = getEllipseCoordinates(svgEllipse)
        TkcOval.new(canvas, coordinates)
    end
end

class TextBuilder
    def svgTag();  return 'text';  end

    def newThing(canvas, svgText)
        coordinates = getTextCoordinates(svgText)
        tx = TkcText.new(canvas, coordinates)
        tx.configure('text', svgText.text())
        tx.configure('anchor', 'center')
    end
end
```

Notice that, because Ruby uses **Dynamic Typing**, the polymorphic array `builders` does not require those two classes to inherit a common base class. They only need methods with the same names and parameters. Ruby calls those methods as if `builders` looked them up via string comparisons.

TODO There are two reasons to inherit in a dynamically typed language—as a Compilable Comment, or to override a method. Refactoring has not yet merged any method to override. But we must first seek something simple to refactor here.

Remember the duplicated "`get`" and "`Coordinates`"? This is where we start to fold them together. First, we move those stray methods inside their objects, and make their names more similar:

```
class EllipseBuilder
    def svgTag();  return 'ellipse';  end

    def getCoordinates(ellipse)
        cx,cy,rx,ry = collectCoordinates(ellipse,
                  ['cx','cy','rx','ry'])

        return [cx - rx, cy - ry, cx + rx, cy + ry]
    end

    def newThing(canvas, svgEllipse)
        coordinates = getCoordinates(svgEllipse)
        TkcOval.new(canvas, coordinates)
    end
end

class TextBuilder
    def svgTag();  return 'text';  end

    def getCoordinates(text)
        return collectCoordinates(text, ['x','y'])
    end

    def newThing(canvas, svgText)
        coordinates = getCoordinates(svgText)
        tx = TkcText.new(canvas, coordinates)
        tx.configure('text', svgText.text())
        tx.configure('anchor', 'center')
    end
```

```
    end
```

Here Comes the Pop

```
    class Builder
        def constructor(canvas, svg, klass)
            return klass.new(canvas, getCoordinates(svg))
        end
    end

    class EllipseBuilder < Builder

    ...
        def newThing(canvas, svg)
            constructor(canvas, svg, TkcOval)
        end
    end

    class TextBuilder < Builder
...
        def newThing(canvas, svg)
            tx = constructor(canvas, svg, TkcText)
            tx.configure('text', svg.text())
            tx.configure('anchor', 'center')
        end
    end
```

What just happened? In testable steps (not shown, but available online), we created a new class called `Builder`, and set the two `*Builders` to inherit it. Then we gave it a method to construct the `Tkc*` item, and called this method from the two `newThing()` methods. Try the refactors in that order.

Many elements of the "UML Sequence Diagram", on the facing page, have the same name. They represent behavioral duplication, which this design folded together.

Curiously, references to the classes `TkcOval` and `TkcText` pass into the `constructor()` method. Programmers only familiar with **Static Typing** languages might not recognize how lightweight the Prototype Pattern becomes in a language with Dynamic Typing, where classes are also objects. Following that pattern, in C++, we would pass a `TkcText` object into `constructor(…TkcItem &anItem)`, and call `anItem.clone()` to obtain a copy of the object that we can use (if a hypothetical C++ version of TkCanvas provided such methods). Ruby classes are themselves objects, so we can pass a reference to the class, instead, and call `klass.new()`. Ruby builds the Prototype Pattern into the language.

Alternately, in C++ we could write a template and specialize it, for each class type we need. This would also add a conceptual burden. Our industry may someday realize that dynamic language techniques enable GUIs better than static ones. Many GUI Toolkits written in Static Typing languages certainly bend over backwards to provide dynamic features.

For each ellipse or text in the SVG, `putSvgIntoCanvas()` calls one or another version of `newThing()`. That passes either `TkcOval` or `TkcText` into `constructor()`. This creates any generic `TkCanvas` item, and calls `getCoordinates()`, virtually, to parse the SVG coordinates and convert them to TkCanvas-style coordinates:

Our new design uses the Abstract Template Pattern. It does common things in a base class method, which calls methods in derived classes to provide specific behaviors.

Your project might not arrive at the same design. When you write a program that parses SVG, or uses `TkCanvas`, or both, a similar or different construction system might emerge. Our new design fits this Case Study's libraries, language, requirements, and refactoring sequence.

These refactors folded duplication together when its count reached two. Not one, or three, or a higher number. When you add the next ability of the same kind, the odds increase for it to reuse an abstraction.

Open Closed Principle

Graphs have nodes, text, and lines ("edges"), at least. Let's add a graph with an edge between its nodes:

```
def test_twoNodesWithAnEdge()
    svg = generateSvg(
        'digraph aGraph { a [label=""]; ' +
            'b [label=""];' +
            'a -> b [dir=none]; }')

    putSvgIntoCanvas(svg, @canvas)
    all = @canvas.find_all()
    assert_equal(all.size(), 3)
    assert_equal(all[0].class.name(), 'TkcOval')
    assert_equal(all[1].class.name(), 'TkcOval')
    assert_equal(all[2].class.name(), 'TkcPolygon')
    maybeMainloop(true)
end
```

Our test cases all look similar. Cloning and modifying test cases helps us write them quickly, helps us start with a passing case before modifying it, and helps us spot reused lines that are good candidates for promotion into test fixtures. But we should not shove everything possible into fixtures. Tests help document code, so their control flow must be obvious.

The new DOT line "`a -> b [dir=none];`" draws the edge. The "`[dir=none]`" attribute prevents `dot` from placing an arrowhead on the end of the line. That would push another graphic element (a little triangle) into the SVG output, and we are forcing our SVG interpreter to learn only one new element at a time.

The test then requests that the canvas have `3` elements, and the last one is a `TkcPolygon`. When we run the tests, the new assertion fails, and Ruby::Unit terminates this test case. The fixture `maybeMainloop(true)` would display the canvas if the test didn't fail. As soon as it passes, our resident acolyte will check its output.

However, the latest version "`graph.svg`" contains a line with some mysteries in it:

```
<path style="fill:none;stroke:black;"
  d="M42,63C42,78 42,97 42,111"/>
```

(To read the file `graph.svg`, put a temporary `exit(0)` in that test case to prevent the next test case from overwriting it. Run the tests, inspect that file, and maybe make an example of it. The most recently written test is not always the last one that the test runner runs.)

The `style` attribute looks straightforward, and we'll get to it eventually, but the `d` attribute has some mysterious codes mixed in with its coordinates.

To decipher these codes, we will do something that goes against all our training as engineers; we will read the SVG documentation. It tells us that path `d`ata work as **Turtle Graphics**; the `M` means "move to", and the `C` means subsequent points anchor a curve.

Now we need code that parses a string and creates Ruby variables. That last paragraph indicates a small part of the SVG documentation now should become our specification.

Suspend the last test (the one that gave us the file "graph.svg"):

```
    def pest_twoNodesWithAnEdge()
…
    end
```

Changing the name of the test case lets the Test Collector skip it. The Collector does not test pests.

Now write a Child Test to convert SVG specification as readable code:

```
    def test_parsePath()
        d     = "M42,63C42,78 42,97 42,111"
        curve = parsePath(d)
        assert_equal([[42,63],[42,78], [42,97], [42,111]], curve)
    end
```

Our first variable's name is "d", matching the SVG attributes. The principle "the same thing has the same name" can be more important than the principle "name things with complete, pronounceable words". Code self-documents when its ubiquitous language cites well-known, published recommendations.

The SVG committee broke the "complete words" principle when it named an attribute "d". Typing all of "data" would doubtless have caused them stress. If we follow the "same name" principle, then the SVG Recommendation itself documents our d. The "same name" principle is more important when it helps detect duplication.

Operating the test-first cycle soon gives us a new function:

```
def parsePath(d)
    d      =~ /M(.*)C(.*)/
    moveTo = $1
    curve  = $2
    moveTo = moveTo.split(/,/)

    #  convert an array of strings into an array of integers

    moveTo = moveTo.collect{ |a| a.to_i() }
    curve  = curve.split(/ /)

    #  convert an array of strings into an array of arrays of integers

    curve = curve.collect{ |a|
        x,y = a.split(/,/)
        [x.to_i(), y.to_i()]
        }
    curve.insert(0, moveTo)
    return curve
end
```

That new function, parsePath(), uses more of those pesky Ruby idioms. We censor the refactors that Ruby experts could otherwise use to squeeze its line count down further (partly because I don't know them). For those of you who already find the function opaque, the test makes good documentation. That's why the name of the test contains the name of the function. Give similar things similar names.

The new function uses a regular expression, /M(.*)C(.*)/, which assumes a leading M, then a C. The SVG documentation recommends L codes for straight lines, and permits letter codes in

any order. This function will break, and not cleanly, when our Representation Layer generates straight paths, or paths containing more `M` codes than the first one. `dot` does not generate those codes, so we might add "find a reason to parse straight lines in paths" to our do-list. We will get around to it shortly after certain very warm places freeze over.

> You (Still) Aren't Gonna Need It! The user wants `dot` graphs in a canvas, not the complete SVG Recommendation in a canvas. Our code is clean and simple, but only supports part of the recommendations.

Now we return to the function `pest_twoNodesWithAnEdge()`, switch it to "`test_`", and get it to fail again. The first failing assertion checks how many items are in the canvas. Fix it by going to `putSvgIntoCanvas()` and adding brute-force code to create a polygon:

```
def putSvgIntoCanvas(svg, canvas)
    doc              = Document.new(svg)
    anEllipseBuilder = EllipseBuilder.new()
    aTextBuilder     = TextBuilder.new()

    builders = [anEllipseBuilder, aTextBuilder]

    builders.each() do |aBuilder|
        XPath.each(doc, '/svg/g/g/' + aBuilder.svgTag()) do
            |svgThing|
            aBuilder.newThing(canvas, svgThing)
        end
    end

    XPath.each(doc, '/svg/g/g/path') do
        poly = TkcPolygon.new(canvas, [10, 10, 20, 20])
        poly.configure('outline', 'black')
        poly.configure('width', 3)
    end
end
```

As usual, we must defer some lofty goal. We will get to the Open Closed Principle as soon as we have passing tests, and a design to clean up.

The new `TkcPolygon` uses bogus coordinates, of course. Fix that by adding to the test statements that get the correct ones, and then assert that our new `TkcPolygon` follows them:

```
    def test_twoNodesWithAnEdge()
        svg = generateSvg(
            'digraph aGraph { a [label=""]; ' +
                'b [label=""];' +
                'a -> b [dir=none]; }')

        putSvgIntoCanvas(svg, @canvas)
        # exit(0)
        all = @canvas.find_all()
        assert_equal(all.size(), 3)
        assert_equal(all[0].class.name(), 'TkcOval')
        assert_equal(all[1].class.name(), 'TkcOval')
        assert_equal(all[2].class.name(), 'TkcPolygon')

        doc     = Document.new(svg)
        d       = XPath.first(doc, '/svg/g/g/path/@d').to_s()
        curve   = parsePath(d)
        curve   = curve.flatten()
```

```
        polygon = all[2]
        coords  = polygon.coords()
        assert_equal(curve, coords)

        maybeMainloop(true)
    end
```

An XPath of `'/svg/g/g/path/@d'` finds every attribute in an XML source whose path matches, and collects all their values. Production code would need a better path, or would need to call `.each()` to iterate through all of them. But the test code can assume there's only one, and can use `.first()` to get it.

Before I added the line `curve = curve.flatten()`, that test failed.

Fail for the Correct Reason

Our SVG reader, `parsePath()`, returns a 2-dimension coordinate array: `[[42, 63], [42, 78], [42, 97], [42, 111]]`. TkCanvas uses flat, 1-dimension arrays: `[42, 63, 42, 78, 42, 97, 42, 111]`. This illustrates the importance of self-documenting assertions. A TkCanvas supports the same coordinate format for all the different `TkcItem` shapes.

Experiments with this test as it fails, observing the complete diagnostic from `assert_equal()`, teach how to adjust the output array format's to match the control array's format. When the test failed for the wrong reason, the diagnostics reported this:

```
Failure!!!
test_twoNodesWithAnEdge(TestGrapher) [sample.rb:195]:
<[[42, 63], [42, 78], [42, 97], [42, 111]]> expected but was
<[10.0, 10.0, 20.0, 20.0]>
```

The two compared arrays have different counts and dimensions. After we fix `curve`, by adding `curve = curve.flatten()` to the test, the diagnostic for the "correct" failure is this:

```
Failure!!!
test_twoNodesWithAnEdge(TestGrapher) [sample.rb:200]:
<[42, 63, 42, 78, 42, 97, 42, 111]> expected but was
<[10.0, 10.0, 20.0, 20.0]>
```

The compared arrays agree in count and dimensions. Future upgrades might move the fix inside `parsePath()`.

First we must make the Simplest Possible Fix to get the test to pass:

```
def putSvgIntoCanvas(svg, canvas)
    doc              = Document.new(svg)
    anEllipseBuilder = EllipseBuilder.new()
    aTextBuilder     = TextBuilder.new()

    builders = [anEllipseBuilder, aTextBuilder]

    builders.each() do |aBuilder|
        XPath.each(doc, '/svg/g/g/' + aBuilder.svgTag()) do
            |svgThing|
            aBuilder.newThing(canvas, svgThing)
        end
    end

    XPath.each(doc, '/svg/g/g/path') do
        d     = path.attributes['d']
```

```
        curve = parsePath(d)
        curve = curve.flatten()
        poly  = TkcPolygon.new(canvas, curve)
        poly.configure('outline', 'black')
        poly.configure('width', 3)
    end
end
```

If we call `maybeMainloop(true)` at the end of the test, we see two ovals and a line between them.

Visually inspecting the output reveals the line cleanly starts at the bottom of one oval and ends at the top of the other. (Our source, at this point, also shows leftover canvases from other tests. More on those soon.) If the line ends missed the ovals, due to some math error, we'd rapidly spot that, add a test for it, and fix it. If various lines often overlapped, these and future tests would get more assertions to check the geometry. Frequent visual inspection reveals what kinds of assertions we need.

The function `putSvgIntoCanvas()` offends the Simplicity Principles *Duplicate No Behavior* and *Minimize Methods*. It has two similar systems to render SVG items; they give it a line count approaching 20.

If we want to reuse the `Builder` abstraction, we have no descendent qualified to create polygons. And attentive readers have learned we won't get there by brute-force coding.

The simplest thing a new `*Builder` class could do, after nothing, is return the name of the SVG type it builds. So we create a trivial class, and then put its single method to use:

```
class PolygonBuilder
    def svgTag();  return 'path'; end
end

def putSvgIntoCanvas(svg, canvas)
    doc = Document.new(svg)
    anEllipseBuilder = EllipseBuilder.new()
    aTextBuilder = TextBuilder.new()

    builders = [anEllipseBuilder, aTextBuilder]

    builders.each() do |aBuilder|
          XPath.each(doc, '/svg/g/g/' + aBuilder.svgTag()) do
                |svgThing|
                aBuilder.newThing(canvas, svgThing)
            end
        end

    aPolygonBuilder = PolygonBuilder.new()

   XPath.each(doc, '/svg/g/g/' + aPolygonBuilder.svgTag()) do
        |path|

        d = path.attributes['d']
        curve = parsePath(d)
        curve = curve.flatten()

        poly = TkcPolygon.new(canvas, curve)
        poly.configure('outline', 'black')
        poly.configure('width', 3)
```

```
    end
end
```

As usual, the design must get a little worse, briefly, before it can get better. And, as usual, the tests don't care; they still pass.

Now, one at a time, we write each method in `PolygonBuilder` that other classes derived from `Builder` have, and we make sure they use the same interface. The ideal, again, is "make them similar, then make them the same, then merge them."

If we weren't taking such small steps, we could write a complete `PolygonBuilder` class, derive it from `Builder`, make `putSvgIntoCanvas()` call it polymorphically, and only then test. It would probably work.

We do this incrementally to prevent mistakes, and to learn. Polygons might require some special behaviors that prevent their class from merging into other shapes' classes. It's best we learn that in small steps, by making the classes similar, before taking the big step and merging the classes.

If this were a static language, to check the interfaces we would inherit an abstract base class. (But static languages still require incremental refactors.) Here in a dynamic language, these classes inherit only to extend the behavior of `constructor()`. Dynamic languages force us to inspect our classes and ensure their methods have same names, arguments, and return types. Before actually calling the class object through a polymorphic interface, we can only visually inspect the methods to ensure they obey the interface.

The result of several such refactors, and their tests, is this:

```
class PolygonBuilder < Builder
    def svgTag();  return 'path';  end

    def getCoordinates(path)
        d = path.attributes['d']
        moveTo, curve = parsePath(d)
        curve = curve.insert(0, moveTo)
        return curve.flatten()
    end

    def newThing(canvas, svg)
        poly = constructor(canvas, svg, TkcPolygon)
        poly.configure('outline', 'black')
        poly.configure('width', 3)
    end
end

def putSvgIntoCanvas(svg, canvas)
    doc = Document.new(svg)
    anEllipseBuilder = EllipseBuilder.new()
    aTextBuilder = TextBuilder.new()

    builders = [anEllipseBuilder, aTextBuilder]

    builders.each() do |aBuilder|
        XPath.each(doc, '/svg/g/g/' + aBuilder.svgTag()) do
            |svgThing|
            aBuilder.newThing(canvas, svgThing)
        end
    end

    aPolygonBuilder = PolygonBuilder.new()
```

```
    XPath.each(doc, '/svg/g/g/' + aPolygonBuilder.svgTag()) do
         |path|
         aPolygonBuilder.newThing(canvas, path)
     end
end
```

We can now inspect that `PolygonBuilder` looks the same as others in the `*Builder` series, so the same code can now call it. All that refactoring permitted the final refactor to require only a few edits. Moving the `PolygonBuilder.new()` into the list of flyweights, and removing those last four lines, is a brief formality that finally reduces the line count:

```
def putSvgIntoCanvas(svg, canvas)
    doc = Document.new(svg)

    builders = [
        EllipseBuilder.new(),
        TextBuilder.new(),
        PolygonBuilder.new(),
        ]

    builders.each() do |aBuilder|
        XPath.each(doc, '/svg/g/g/' + aBuilder.svgTag()) do
            |svgThing|
            aBuilder.newThing(canvas, svgThing)
        end
    end
end
```

Traditionally, the most common “organic design” lacks tests and aggressive refactoring. It leads to long, run-on functions. If we had refactored nothing, and only added tests and code, by now our function `putSvgIntoCanvas()` would be very long, refactoring it would be rough, and new abilities would be hard to add.

Rest State

When a good design can’t productively refactor any more, call it a “rest state”. The physics metaphor is electrons flying around atomic nuclei. Energy (here programmers adding features) pushes the electrons into higher orbits with more complex shapes. When the electrons return energy (refactoring), they fall into lower and simpler orbits. Their lowest orbit (for a given ambient temperature) is their “rest state”.

The “rest state” for `putSvgIntoCanvas()` is 4 lines with behavior, 9 total. When we added a feature it puffed up; then we refactored it back down. Unlike physics, this “rest state” is now more likely to persist. Because we refactored our design while adding similar features, new features of the same type should not require more refactoring.

Conclusion

We selected a few very high-level components (GraphViz, TkCanvas, REXML, and Ruby::Unit), and integrated them with GUI code developed test-first. We rapidly built a robust module that a program can call with a very rich and completely standardized interface. And the module’s test harness grew with it. This increases confidence (and sample code) to help upgrade or reuse the module.

We were rapid because the TkCanvas retains graphic commands as objects. Any property that production code can Set, test code can Get. Without that benefit, this project would have

been much more complex. The earliest phase would have mixed growing a design with growing fixtures that retain graphic commands, so tests could meaningfully constrain them. Only after such a difficult effort (and difficult narration) could test-first have lead to flow.

Presently, anything our clients can express in DOT notation, they can view in a Canvas, rendered not as raw pixels but as intrinsic objects.

Except for arrowheads. Or the SVG "`style`" attribute. I forgot them! We must sprint before concluding this chapter…

First, write a *Temporary Visual Inspection* test that draws a big full graph, to see what's missing. We want a big graph, which someone else defined, to increase the odds that it looks bad. If we wrote it ourselves, it might carry the same assumptions as our simpler graphs, decreasing the odds it looks bad.

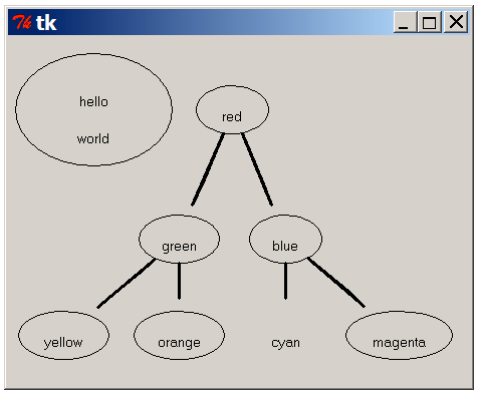
I found one of `dot`'s sample files with many attributes in it, and put it inside a new test case. I made it **`bold`** because it's new, relative to the test and production fixtures which we have seen before:

```
    def test_style()
        svg = generateSvg(
            'digraph G {
                xyz [label = "hello\nworld",
                    color="slate_blue",
                    fontsize=24,
                    fontname="Palatino-Italic",
                    style=filled,
                    fontcolor="hot pink"];
                node [style=filled];
                red [color=red];
                green [color=green];
                blue [color=blue,fontcolor=black];
                cyan [color=cyan,shape=box];
                magenta [color=magenta];
                yellow [color=yellow];
                orange [color=orange];
                red -> green [color=brown];
                red -> blue [color=purple];
                blue -> cyan [color=blue];
                blue -> magenta;
                green -> yellow;
                green -> orange; }'
            )

        putSvgIntoCanvas(svg, @canvas)
        maybeMainloop(true)
    end
```

(And note that `Tk.restart()` can also stop and start our event loop…)

As this module grows, we should incorporate more GraphViz sample files into our tests. Perhaps an Acceptance Test should read a list of them, chart each one, and perform sanity checks on each one. We had to start simple, and GraphViz's authors wrote sample files to illustrate complex displays, so until now no test has used a sample file for input.

When `maybeMainloop(true)` raises the current window, nothing has any style:



Arrowheads are missing, the `cyan` node's `shape` is not a `box`, nothing has a color, the lines are too thick, and "hello world" should use a larger font. (Write that sample DOT notation to a file, and use `dot` on a command line to output an alternate rendering, such as a PNG file, to compare its correct output to that canvas.)

If this project had a Customer Team, they would review the difference between our SVG Canvas and `dot`'s own PNG output, to declare which missing features are more important, and which to let slide. Maybe the users won't need arrowheads, but will need colors.

We will pretend our Customer Team prioritized arrowheads, then boxes, then colors. And we pretend they pushed thin lines and fonts out of this iteration, so we should neither do them nor think about them.

To implement arrowheads and such, we need to see how the SVG string represented them. So change the test, to visually inspect the current SVG string:

```
        putSvgIntoCanvas(svg, @canvas)
        puts(svg)
        maybeMainloop()
```

Here's a sample of relevant SVG. I marked features to implement in **`bold`**:

```
<g id="node5" class="node">
    <title>cyan</title>
    <polygon style="fill:cyan;stroke:cyan;"
             points="312,274 239,274 239,322 312,322 312,274" />
    <text text-anchor="middle" x="276" y="305">cyan</text>
</g>
<g id="edge6" class="edge">
    <title>blue-&gt;cyan</title>
    <path style="fill:none;stroke:blue;"
          d="M276,226C276,237 276,249 276,261" />
    <polygon style="fill:blue;stroke:blue;"
             points="280,261 276,274 273,261 280,261" />
</g>
<g id="node6" class="node">
    <title>magenta</title>
    <ellipse cx="389" cy="298" rx="53" ry="24"
             style="fill:magenta;stroke:magenta;" />
    <text text-anchor="middle" x="389" y="305">magenta</text>
</g>
```

Both arrowheads and boxes use the `<polygon>` tag (not the SVG `<path>` tag). Note that `node5`, the "`cyan`" node that should be `box shaped`, has a `<polygon>` tag. And note that `edge6` is just one of many that should have an arrowhead; it also has a `<polygon>` tag.

SVG supports several more primitives, including `<rect>` for rectangles. If your Representation Layer (`dot`) does not request them, then you aren't gonna need them. If your Representation Layer ever extends into these features, only then would you match its abilities in your GUI Layer.

> You are going to need features. If you were writing the Representation Layer instead of using one off the shelf, you would not pack it full of features, and only then work on the GUI Layer to match it. Both modules must grow together. This is why programmers own tasks, not modules. Any programmer on your team could add, for example, `<rect>` to an SVG generator module, then upgrade "your" canvas to display it.

Arrowheads

The Customer Team says we are going to need them. Their SVG looks like this:

```
<polygon style="fill:blue;stroke:blue;"
         points="280,261 276,274 273,261 280,261" />
```

We can guess that `putSvgIntoCanvas()` will soon interpret the `polygon` tag, and it will need the ability to parse that `points` attribute string. So we skip ahead to the low-level part, a new function called `parsePoints()`.

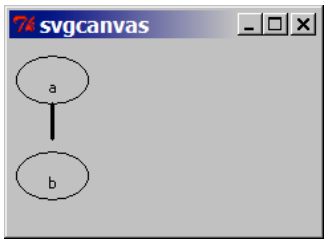
Copy the `points` attribute into a new test, and pass it:

```
    def test_parsePoints()
        points = "312,274 239,274 239,322 312,322 312,274"
        points = parsePoints(points)

       assert_equal([312,274,239,274,239,322,312,322,312,274],
                 points)
    end

…
    def parsePoints(pointString)
        points = pointString.split(/ /)
        points = points.join(',')
        points = points.split(/,/)
        points = points.collect{|x| x.to_i() }
        return points
    end
```

Both paths and polygons in SVG render in a TkCanvas as `TkcPolygons`. Our big "Smoke Test" created many `polygons`, and a little smoke. A test to drive arrowhead development needs a smaller graph, without competing `polygons`. We need a case to focus on a single `TkcPolygon`, without wondering where in a list of elements it should appear. So we want a simple graph with two ellipses, a line between them, and an arrowhead on the end of the line:

```
        def test_arrowhead()
            svg = generateSvg('digraph aGraph { a -> b }')
            putSvgIntoCanvas(svg, @canvas)
            maybeMainloop(true)
            system('dot -Tpng graph.dot -o test_arrowhead.png')
        end
```

That case emits two *Temporary Visual Inspections*. The first uses the traditional `maybeMainloop(true)` to reveal the bug we target—the end of a line that misses its arrowhead:



The second *Temporary Visual Inspection* intercepts our current "`graph.dot`" file (before the next case writes on it) and converts it into this, revealing the arrowhead goal:

(That `digraph` looks familiar. This iteration has come full circle!)

We need a test to force `putSvgIntoCanvas()` to contain two ovals, two text items, a line, and an arrowhead. The `canvas.find_all()` property stores elements in the order that `putSvgIntoCanvas()` created them. Our arrowhead will be the sixth item in the canvas. (When our new test hard-codes `[5]`, to index the sixth item, any upgrades that rearrange the canvas elements will break this test, even while their output appears correct. See page 113 for an example of a code fix breaking hyperactive assertions.) Running this test as often as possible, to prevent sloppy refactors and upgrades, converts this risk into more rapid development:

```
def test_arrowhead()
    svg = generateSvg('digraph aGraph { a -> b }')
    putSvgIntoCanvas(svg, @canvas)

      #   the line from a to b
    line = @canvas.find_all()[4]
    assert_equal('TkcPolygon', line.class.name)

      #   the arrowhead on the end
    aPolygon = @canvas.find_all()[5]
    assert_equal('TkcPolygon', aPolygon.class.name)
    assert_equal(8, aPolygon.coords.length)

      #   no more elements
    assert_equal(6, @canvas.find_all().length)
    maybeMainloop(true)
end
```

A `PolygonBuilder` class will create our arrowhead. Unfortunately, a class called "`PolygonBuilder`" already exists, and it builds paths, so we will quietly rename it "`PathBuilder`" as we go. All the `*Builder` classes now have the SVG tags they support in their names. Until now the code broke the rule "Same thing same name".

The passing code:

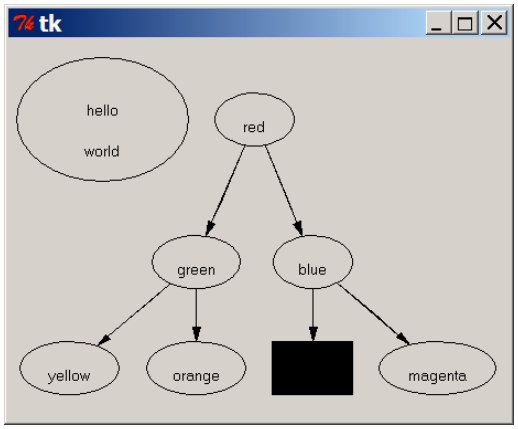
```
class PolygonBuilder < Builder
    def svgTag();  return 'polygon'; end
```

```
        def getCoordinates(path)
            points = path.attributes['points']
            return parsePoints(points)
        end

        def newThing(canvas, svg)
            poly = constructor(canvas, svg, TkcPolygon)
            poly.configure('outline', 'black')
            poly.configure('width', 1)
        end
    end
...
        builders = [
            EllipseBuilder.new(),
            TextBuilder.new(),
            PathBuilder.new(),
            PolygonBuilder.new(),
            ]
```

We draw the ellipse and polygon elements first, so they paint "under" the text.

Boxes

Returning to our big viewable test, it now shows more polygons:



We have arrowheads! And a box!

(`dot` converts arrowhead commands into little triangles. It skewed the lower left and lower right ones a little. This iteration cannot address those esthetics!)

Style

The box is `black`, not `cyan`, to remind us that colors come next. We must extract them from the `style` attribute. It contains property names and colors, between `:` and `;` punctuation:

```
<g id="node6" class="node">
  <title>magenta</title>
  <ellipse cx="389" cy="298" rx="53" ry="24"
           style="fill:magenta;stroke:magenta;" />
  <text text-anchor="middle" x="389" y="305">magenta</text>
```

```
</g>
```

Tk, GraphViz, and SVG all agree on a set of colors expressed as words in English, such as "`magenta`", and less convenient RGB values, such as "`255, 0, 127`". Without the small mercy of a common verbose color table, we would need a conversion table. Tip: If your project needs such a table, borrow the one inside the Tk source, instead of writing it from scratch!

Most languages support lists of keys and values as a "hash" or "map". So before worrying about how to push `style` into our canvas, we can first write a Child Test and a function that changes the representation from a delimited string to something more convenient:

```
    def test_parseStyle()
        style = "font-family:Courier;font-size:24.00;fill:hotpink;"
        style = parseStyle(style)

        expect = {
            'font-family' => 'Courier',
            'font-size' => '24.00',
            'fill' => 'hotpink',
            }

        assert_equal(expect, style)
    end

...
def parseStyle(style)
    keysAndValues = style.split(/;/)
    keysAndValues = keysAndValues.collect{|x|  x.split(/:/)  }
    style = {}
    keysAndValues.each{|x|  style[x[0]] = x[1]  }
    return style
end
```

Like Perl, Ruby can make statements that convert delimited strings into data structures easier to type than to read.

Now let's see if we can get the `cyan` box to appear Cyan. The easiest way to write this test is clone `test_arrowheads()` and change the graph nodes around in the DOT notation:

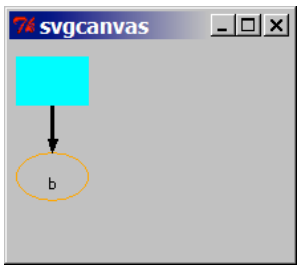
```
        def test_arrowheadsAndStyle()
            svg = generateSvg('digraph aGraph {
                            a [color=cyan, shape=box, style=filled];
                            b [color=orange];
                            a -> b }')
            putSvgIntoCanvas(svg, @canvas)


              #   the line from a to b
            aLine = @canvas.find_all[3]
            assert_equal('TkcPolygon', aLine.class.name)


              #   the arrowhead on the end
            aPolygon = @canvas.find_all[5]
            assert_equal('TkcPolygon', aPolygon.class.name)
            assert_equal(8, aPolygon.coords.length)
            assert_equal(6, @canvas.find_all.length)


              #   the second element is a box (TkcPolygon)
            aPolygon = @canvas.find_all[4]
            assert_equal('TkcPolygon', aPolygon.class.name)
```

```
            # its outline color is cyan
            assert_equal('cyan', aPolygon.cget('outline'))

            # the first element is an ellipse
            anEllipse = @canvas.find_all[0]
            assert_equal('TkcOval', anEllipse.class.name)

            # its outline color is orange
            assert_equal('orange', anEllipse.cget('outline'))

            maybeMainloop(true) # and we can see their text
        end
```

Here's just one of the edits to support `style`:

```
class EllipseBuilder < Builder
...
    def newThing(canvas, svg)
        oval = constructor(canvas, svg, TkcOval)
        style = parseStyle(svg.attributes['style'])

       if ! style['stroke'].nil? and
            style['stroke'] != 'none' then
            oval.configure('outline', style['stroke'])
        end

        if ! style['fill'].nil? then
            if style['fill'] == 'none' then
                oval.configure('fill', '')
            else
                oval.configure('fill', style['fill'])
            end
        end
    end
end
```

Those new statements look very … traditional. The poor design and risk for bugs call for more refactoring. But the Test-First Programming cycle forbids refactors until all tests pass, and `maybeMainloop(true)` reminds us the code ability, and its tests, is not finished yet:



Our *Temporary Visual Inspection* did its job. The assertions all passed, but the appearance is at fault. More assertions, like our current ones, would not help. If we checked the first `TkcText` item exists, and contains "a", the tests would pass. However, the current stack of graphic primitives is "`TkcOval`, `TkcText`, `TkcText`, `TkcPolygon`, `TkcPolygon`, `TkcPolygon`", and items higher in the list overwrite lower ones. `TkcText` writes an "a", then the cyan `TkcPolygon`

box blots it out. A new kind of assertion, to check the Z-order, could catch it. We will skip ahead to the fix.

Build `TkcPolygons` first. Re-arrange the list of builders, from…

```
    builders = [
       EllipseBuilder.new(),
       TextBuilder.new(),
       PathBuilder.new(),
       PolygonBuilder.new(),
          ]
```

…to:

```
    builders = [
       EllipseBuilder.new(),
       PolygonBuilder.new(),  #  behind the TkcText items
       TextBuilder.new(),
       PathBuilder.new(),
       ]
```

That fix breaks every test which uses `@canvas.find_all[x]`, where *x* is a number based on an assumption regarding the order of items in the `builders` list. Our hyperactive assertions have come home to roost!

Change…

```
                #  the line from a to b
              line = @canvas.find_all[4]
              assert_equal('TkcPolygon', line.class.name)
```

…to:

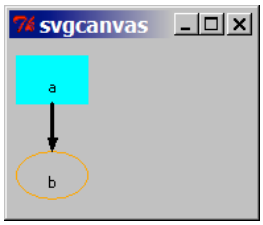
```
                #  the line from a to b
              line = @canvas.find_all[5]
              assert_equal('TkcPolygon', line.class.name)
```

A few similar changes ripple through the tests.

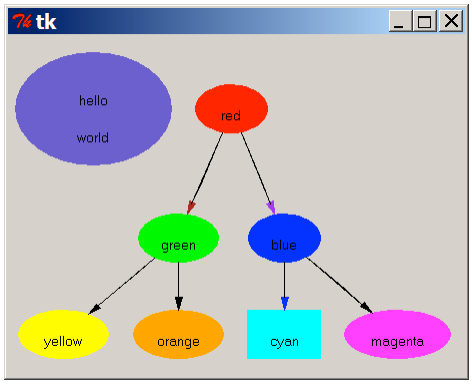
Per *Fuzzy Matches*, on page 37, one might introduce fuzziness when assertions are hyperactive. For example, we could replace `find_all` with a new method that seeks `TkcPolygons`, anywhere in a canvas. However, in this specific situation, the same assertions are now *less* hyperactive, because they constrain a more meaningful and less arbitrary sequence of `builders`. Our tests now influence which canvas items draw first, and which draw last.

(A better fix should draw items in SVG's native document order. You are not going to need an SVG Canvas that draws any SVG except that produced by `dot`, which uses other rules to prevent overlaps.)

After we fix some more assertions' indices, `maybeMainloop(true)` confirms the new situation:



When we activate `test_style()`'s `maybeMainloop(true)`, it shows many more colors (which, in print, appear as shades of grey):



Emergently, I'm surprised some of the arrowheads are colored too (but you can't see that). Credit replacing duplications with abstractions (in both our code and `dot`'s innards). However, our canvas should also color lines and text. A little more duplication removal should emerge these effects.

Our code translates between SVG's `style` attributes and Tk's `configure` variables. The simplest refactor, to make the translator reusable, would be a method such as `configureStyle(element, style, 'fill')`. It will pull the `'fill'` value out of the `style` map, and push it into the configurations of `element`.

But don't fold too much duplication. The string `'fill'` is really two strings that look the same. One belongs to SVG; the other to TkCanvas. (By contrast, `'outline'` and `'stroke'` are already different, and won't confuse us.)

Folding together the correct amount of duplication into a translation method produces this:

```
    def configureStyle(element, style, configuration, styleKey)
        styleValue = style[styleKey]

        if ! styleValue.nil? then
            styleValue = ''  if styleValue == 'none'
            element.configure(configuration, styleValue)
        end
    end

    class EllipseBuilder < Builder
...
        def newThing(canvas, svg)
            element = constructor(canvas, svg, TkcOval)
            style = parseStyle(svg.attributes['style'])
            configureStyle(element, style, 'outline', 'stroke')
            configureStyle(element, style, 'fill', 'fill')
        end
    end
```

And that's it. We can translate an SVG `'none'` into a TkCanvas `''`, and `'stroke'` into `'outline'`. This early in the `style` project, putting similar things right next to each other creates a little table; this may become the seat of more refactors. Further tests and refactors will reuse `configureStyle()` in other `*Builders`, and will make `configureStyle()` a member of `Builder`. That will permit us to call `parseStyle()` in only one place—inside `constructor()`—so we can make `element` and `style` into class members, and eliminate their duplications as arguments.

Long before finishing all possible `dot` output, such as fonts, and before our design finishes changing, our code is ready to extend into other SVG features. And we only need to take these

features one at a time. We are not selling a complete “SVG Canvas Library” that must appeal to programmers with arbitrarily complex SVG. Our module grows together with its peer modules, focusing only on features our Customer Team requested.

As our conclusion, we simulated changing requirements late in a project. Then we quickly extended the object model, the way it designed to extend. We added a new, orthogonal abstraction, without destabilizing anything. And each new test learned a new way to *Query Visual Appearances*.

Internally, `dot` provides SVG with much more information than we use, and TkCanvas elements can store and process many more data than we currently push into them. Our next experiment provides an interactive client application for this module, and integrates more of `dot`’s output into the Canvas items.

Chapter 6: Family Tree

The last Case Study produced a canvas with graph nodes and edges drawn on it. That chapter ignored user interactions—no selection emphasis, keyboard navigation, or editing. Its Case Study revealed test-first coding, refactoring to create a design, and an easy example of the Principles *Just Another Library* and *Event Queue Regulation*.

Our mouse now hovers over that canvas, poised to click on a node, to see what happens. User interactions add another layer of complexity to a test-first cycle, generating more test fixtures. This project uses many *Temporary Interactive Tests* to *Simulate* diverse kinds of *User Input*.

We pretend our users want to draw a family "tree" (which some cultures weave into quite complex cyclic graphs). That requirement leads to the low-level features this Case Study develops. We will enable users to manipulate nodes, draw links between nodes, save new DOT files, and change node text:

The sequence of activities:

- Page 116: Bootstrapping interaction.
- 120: Accurately simulate user inputs.
- 134: The user can edit a node to change its text.
- 138: The canvas can save a new DOT file.
- 146: The user can drag arrows between nodes.

The last Case Study selected TkCanvas because it retains graphics commands as objects. If a canvas converted commands to pixels, it would discard information about those commands, complicating test cases. That Case Study also selected TkCanvas to assist this one. Retaining graphic commands as objects permits a canvas to dispatch user interactions directly to those objects. Put another way, if a program sends graphic commands to a canvas that converts them into a raster, the program must correlate mouse locations with graphic element locations at click time. TkCanvas binds event handlers to their individual items, making these features easy.

Bootstrapping Interaction

Our first *Temporary Interactive Test* lets us manually experiment with program behavior at click time. Such tests work by instrumenting an event handler with a trace utility. So our first task is to write a function that traces any object's properties.

Ruby reflects objects' members with the method `public_methods()`. We can wrap that up nicely into a function that `doc`uments any object:

```
def doc(anObject)
    puts(anObject.class.name)

    itsMethods = anObject.public_methods() -
                                    Object.new().public_methods()

    puts(itsMethods.sort())  if !itsMethods.nil?
end
```

The previous Case Study got very far without explaining too much Ruby, so we can risk learning a little more here. `puts()` works like the C Standard Library `puts()`. It writes strings to the standard output stream, then adds linefeeds after each string.

`anObject.public_methods()` returns a list of names of public methods. All Ruby objects inherit a global base object called `Object`, which supplies its own public methods. We don't need `doc()` to report those over and over again, so the subtraction operator, `-`, removes all of `anObject`'s public methods that also appear in a new `Object`'s public methods.

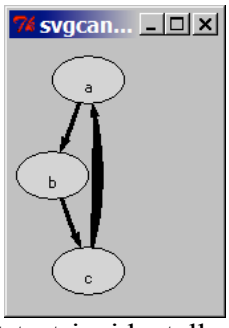
Ruby is generally line-oriented. Lines end logically with a linefeed, not a semi-colon. Some lines would be too long, and must break. Ruby treats trailing binary operators, like the subtraction operator, `-`, as line-continuations. Ruby parses the next line as part of the broken one.

That `doc()` utility generates a short list of its argument's methods. If the list is not `nil`, it sorts and prints each element. You can use it to learn about any Ruby object whose documentation might be … inscrutable.

That trace utility will help research what makes Tk events tick. Our test case will bind a trace statement to a node in a graph, and open the `maybeMainloop(true)` spigot on our event queue:

```
    def test_simulateMouseClick()
        svg = generateSvg('digraph aGraph {
                           node [style=filled];
                           a -> b -> c -> a }')

        putSvgIntoCanvas(svg, @canvas)
        oval = @canvas.find_all[0]
        assert_equal(oval.class.name(), 'TkcOval')

        oval.bind('Button-1') {
            |event|
            doc(event)
            }

        maybeMainloop(true)
    end
```

Enter that, run the tests, and click next to the "a":



That test incidentally revealed a display bug. The last Case Study never wrote DOT files that generated curved lines. A real iteration would have experimented with many more sample DOT files. Curved lines accidentally draw as areas, with a straight line closing their shapes. We will make a note of this bug (and the DOT notation that caused it), and fix it after the current drive.

Clicking a Tk window with the first mouse button (typically the left one) sends an event called `'Button-1'`. Ruby/Tk responds to them with block closures. This statement…

```
oval.bind('Button-1') {
    |event|
    doc(event)
    }
```

…binds the "a" oval to respond to that mouse click by documenting the `event` object it passes. Clicking anywhere else does nothing, but clicking inside the "a" oval writes this to a console:

```
TkEvent::Event
above
borderwidth
char
count
detail
focus
height
...
keycode
keysym
keysym_num

x
x_root
y
y_root
```

The `Event` object has almost 30 properties. We won't need most of them. Input event objects bear a record of the GUI's hardware state at the moment the hardware event arrived. That event may have occurred a few milliseconds (or more) before the event object reached our callback. So GUI Toolkits must pack a snapshot of various conditions into each event object, or provide a method to query them. Otherwise, on a system with few CPU cycles to devote to the GUI, users might perceive mouse clicks "missing" their targets.

Our TkCanvas supports another way to miss these targets. Notice, on the previous page, the DOT command `node [style=filled]`. That instructs `dot` to fill each oval with a default color, grey. Without that style, the TkCanvas would draw ovals as little black rings. Clicking on a ring would trigger an event, but clicking the background color inside a ring would not trigger an event.

The finished program should also trap mouse events on the `TkcText` items themselves. Clicking directly on the `a`, `b`, or `c`, above, won't select the node under them.

Block Closures

Binding event handlers to GUI controls is the heart of GUI architecture.

> The best event handlers are block closures.

Not many languages provide these, including many inexplicably popular languages promoted as designed to support GUIs. To learn about them, add an experiment to our *Temporary Interactive Test*:

```
X = 42

oval.bind('Button-1') {
    |event|
    doc(event)
    puts(x.inspect())
    }
```

Ruby statements between `{` and `}` (or `begin` and `end`) form an object, called a block. Each time that code calls `oval.bind(){…}`, it instantiates a new block object, and passes it into any method to its left. That method can evaluate the block any number of times, and can store it, as a `Proc` object, for later use.

Each time a block instantiates, it links to the current instances of any local variables seen around it. All block evaluations use those variable copies, as if their scope were still active. If a block lives longer than its local method, those variables remain alive too. This valuable technique helps restrict variable scopes, preventing the need to make them into member variables. Block closures give local variables the narrowest possible scope over an indefinite lifespan.

Ruby executes the above statement by calling `bind()`, and passing into it a handle to the block that starts on its right. `bind()` stores the block in the `TkcOval` object's private memory. Control flow then enters `maybeMainloop()`, enters `Tk.mainloop()`, paints the window, and waits for hardware events. When we click on the canvas item, Tk constructs an `event` object, and passes it into our block. It arrives in our `|event|` parameter variable, and we drop it into `doc()` to see its members.

Then the block `inspects x` and prints `42` onto the console, even though `x`'s outer scope ended before we clicked on the `oval`.

Future changes will put something useful inside the block, and will move it into the production code. Long term, despite how useful block closures are, many block closures for GUI event handlers do nothing but delegate to a non-closure method:

```
oval.bind('Button-1') {
    |event|
    importantMethod(event.x, event.y, oval)
    }
```

That permits *Loose User Simulation*, where tests that *Simulate User Input* need only call `importantMethod()` directly, without the extra effort of concocting a fake `TkComm::Event` object and routing it through the `oval` object into its bound block.

Firm Tk User Simulation

The *Temporary Interactive Test* has one more job. We will use it to write a fixture that simulates input events. First, make the test fail because nobody clicks on our `oval`:

```
        def test_simulateMouseClick()
…
            caughtEvent = false

            oval.bind('Button-1') {
                |event|
                # doc(event)
                caughtEvent = true
                }

            assert(caughtEvent)
            maybeMainloop()
        end
```

Now, research how to do something that Tk tutorials don't cover—simulate an input without displaying a window. We could call `.event_generate()`, but that requires a visible window, which requires `Tk.update()` to paint the window. We will see how long we can go with silent tests.

To simulate an input without breaking Tk, think of that `.bind()` statement as a Set, and research the matching Get. We will retrieve the `oval`'s bound methods with `.bindinfo()`, fetch out the `Proc` entry, and call it directly:

```
        def test_simulateMouseClick()
…
            caughtEvent = false

            oval.bind('Button-1') {
                caughtEvent = true
                }

            cb_entry = oval.bindinfo('Button-1')[0][0]
            cb_entry.call()

            assert(caughtEvent)
            maybeMainloop()
        end
```

That test's block forms a closure with `caughtEvent`. Then `.bindinfo()` returns an array of arrays of handlers. We copy out the first one (and we assume our production code will always only create one), and call it. Tk calls our block and toggles `caughtEvent`, then we assert it was toggled.

Although that test passes, this effort is not finished. We speculate that within this iteration we will need to simulate more input events to more canvas items. Sometimes test fixtures emerge, and sometimes we proactively create them. (Especially the ones that satisfy the problems listed on page **Error! Bookmark not defined.**368.)

Convert the useful part into a test fixture:

```
        def click(what, how = 'Button-1')
            cb_entry = what.bindinfo(how)[0][0]
            cb_entry.call()
        end
```

```
        def test_simulateMouseClick()
...
            click(oval)
            assert(caughtEvent)
            maybeMainloop()
        end
```

We can now write test cases that simulate user events. There are ways to configure the Event object's members that we researched. For now we guess the next few features won't require `click()` to transmit any `event` object members.

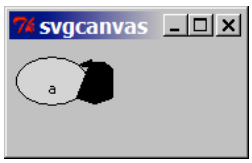
Most projects can take even more shortcuts, and use *Loose User Simulation*. If we bound a block with `oval.bind('Button-1') { foo() }`, we could just test by calling `foo()`. Teams must maintain awareness of these risks and tradeoffs. Our current fixtures trade accurate event simulation for silent test cases that don't flicker their target windows. That decreasing the odds we accidentally notice display bugs.

Goal Stack

While developing the last section, we pushed an item onto our goal stack. Between tasks, we pop all stack items, finish the easy ones, and escalate the hard ones to our team.

This one is easy. Look at the picture on page 118. The lines are thick because they are really filled polygons. The thick crescent on the right should be a thin curved line. This *Temporary Visual Inspection* reproduces the problem:

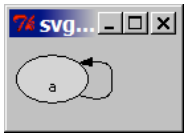
```
        def test_selfReference()
            svg = generateSvg('digraph aGraph {
                                node [style=filled];
                                a -> a }')

            putSvgIntoCanvas(svg, @canvas)
            line = @canvas.find_all[3]
            assert_kind_of(TkcLine, line)
            maybeMainloop(true)
        end
```



A self-referential node demands a curved edge. Ours connects the beginning to the end, and accidentally fills everything in. The fix exposes a small mismatch. `dot` emits SVG using one `path` concept to represent both disconnected lines and filled shapes. Our beautiful `*Builder` design requires a lowly `if` statement to tell the difference:

```
    class PathBuilder < Builder
...
        def newThing(canvas, svg)
            style = parseStyle(svg.attributes['style'])

            if 'none' == style['fill'] then
                poly = constructor(canvas, svg, TkcLine)
```

```
            configureStyle(poly, style, 'fill', 'color')
        else
            poly = constructor(canvas, svg, TkcPolygon)
            configureStyle(poly, style, 'outline', 'stroke')
        end

        poly.configure('smooth', '1')
    end
end
```



That change requires a few test cases to switch their assertions from `TkcPolygon` to `TkcLine`. *Fault Navigation* helps us find and change these without breaking flow.

Clearing our goal stack lets us return to this Case Study's mission. User interaction requires continuously illustrating which object the user is interacting with.

Selection Emphasis

Now that our tests can click on our nodes (first), we need to give users some feedback when they click their nodes. First, take a survey of the properties each `TkcOval` can Get and Set. Find an `oval` in the testage, and temporarily put this line under it:

```
        puts(oval.configinfo().inspect())
```

It reports an info-blitz of options, including these:

```
[ ["activedash", "", "", "", ""],
  ["activefill", "", "", "", ""],
...
  ["fill", "", "", "black", "black"],
...
  ["outline", "", "", "black", "black"],
...
  ["dash", "", "", "", ""],
...
  ["width", "", "", 1.0, 1.0],
...
]
```

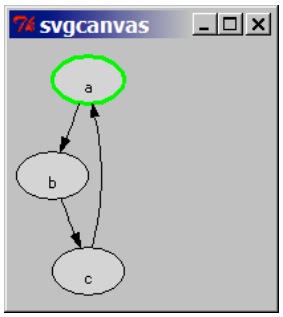
Those are the variables that `.configure()` has been tweaking. `.configinfo()` returns their complete table, so we can read it to learn what options are available. To make a node look selected, we'll change its `'outline'` to `'green'`, meaning "Go", and change its `'width'` to `3`.

This *Temporary Visual Inspection* Learner Test creates a graph, gives one node a callback, simulates clicking on that node, and Gets its configurations with `.cget()`:

```
    def test_selectNode()
        svg = generateSvg('digraph aGraph {
                           node [style=filled];
                           a -> b -> c -> a }')
```

```
            putSvgIntoCanvas(svg, @canvas)
            oval = @canvas.find_all[0]
            assert_equal('TkcOval', oval.class.name())
            assert_equal('black', oval.cget('outline'))
            assert_equal(1.0, oval.cget('width'))

            oval.bind('Button-1') { |event|
                oval.configure('outline', 'green')
                oval.configure('width', 3.0)
                    }
            click(oval)
            assert_equal('green', oval.cget('outline'))
            assert_equal(3.0, oval.cget('width'))
            maybeMainloop(true)
        end
```

That displays this:



Clicking other nodes doesn't select them, so we ought to treat that as a bug and fix it. First, convert this from a Learner Test to a user simulation test by moving that `.bind()` statement (and its block) into the production code:

```
    class EllipseBuilder < Builder
...
        def newThing(canvas, svg)
            element = constructor(canvas, svg, TkcOval)
            style = parseStyle(svg.attributes['style'])
            configureStyle(element, style, 'outline', 'stroke')
            configureStyle(element, style, 'fill', 'fill')

            element.bind('Button-1') { |event|
                element.configure('outline', 'green')
                element.configure('width', 3.0)
                }
        end
    end
```

The test, not shown, now asserts that the first oval's outline is black, and after a click it is thick and green.

The production code implies that clicking on any ellipse gives it a green outline. The code does not remove the green selection emphasis from other ellipses, so clicking many ellipses will select them all. (Also, nodes that are not ellipses won't get the selection emphasis, and ellipses that are not nodes will. Because we are lazy—oops I mean "Agile"—we will only fix these potential issues as we encounter them.)

When we click on the current version of our canvas, we can select more than one oval node, so fix that next.

Add to the test a trap for the bug where selection does not transfer to another clicked ellipse:

```
            oval_b = @canvas.find_all[1]
            assert_equal('TkcOval', oval_b.class.name())
            assert_equal('black',   oval_b.cget('outline'))
            assert_equal(1.0,       oval_b.cget('width'))
            click(oval_b)
            assert_equal('black', oval  .cget('outline'))
            assert_equal(1.0,     oval  .cget('width'))
            assert_equal('green', oval_b.cget('outline'))
            assert_equal(3.0,     oval_b.cget('width'))
```

That forces the system to remember the last selected item, and to restore its format:

```
    class EllipseBuilder < Builder
…
        def newThing(canvas, svg)
            oval = constructor(canvas, svg, TkcOval)
            configureStyle('outline', 'stroke')
            configureStyle('fill', 'fill')

            oval.bind('Button-1') { |event|
                if $lastItem then
                    $lastItem.configure('outline', 'black')
                    $lastItem.configure('width', 1.0)
                end

                oval.configure('outline', 'green')
                oval.configure('width', 3.0)
                $lastItem = element
                }
        end
    end
```

Readers with allergic reactions to the sight of global variables in other languages might now feel mysteriously itchy. Ruby can slip global variables in very quietly. The `$lastItem` is a global, even though it only appears inside one method. It will cause trouble soon. When this code grows an object to wrap the canvas, the global will become its member. But we are not done this feature yet.

> Each time one adds a mouse or keyboard action, one should immediately add the complementing keyboard or mouse action.

In the olden days, computers had no mice. Then, for a while, poorly written applications did not support the mouse correctly. Today, many programs require mouse abuse while neglecting the keyboard.

In this project, most complementing keystrokes should run from a “Context Menu”, which is generally easy to test-first. This Case Study tests mousestrokes first, because they are harder.

Move the code that selects into its own function:

```
        def newThing(canvas, svg)
            element = constructor(canvas, svg, TkcOval)
…
            element.bind('Button-1') {
```

```
                select(element)
                }
        end
...
    def select(element)
        if ! $lastItem.nil? then
            $lastItem.configure('outline', 'black')
            $lastItem.configure('width', 1.0)
        end

        element.configure('outline', 'green')
        element.configure('width', 3.0)
        $lastItem = element
    end
```

In the tests, write a fixture that detects if a canvas item, by index, is selected:

```
        def isSelected(itemIndex)
            item = @canvas.find_all[itemIndex]

            return ( 'green' == item.cget('outline') and
                     3.0     == item.cget('width') )
        end
```

We are ready to test-first again. Tk calls <Tab> keystrokes `'Key-Tab'`, so this new test expects that keystroke to move the selection to the next node:

```
        def test_tab()
            svg = generateSvg('digraph aGraph {
                               node [style=filled];
                               a -> b -> c -> a }')

            putSvgIntoCanvas(svg, @canvas)
            assert(!isSelected(0))
            click(@canvas, 'Key-Tab')
            assert(isSelected(0))
            maybeMainloop()
        end
```

To pass that test, bind a keystroke event handler to the canvas:

```
    def putSvgIntoCanvas(svg, canvas)
...
        canvas.bind('Key-Tab') {
            if canvas.find_all.size() > 0 then
                select(canvas.find_all[0])
            end
            }
    end
```

We expect that test to pass, but it crashes by throwing an exception. Our code does not catch it, so Test::Unit catches it and prints out its stack trace. This includes:

```
NameError: invalid command name `.w00007'
    c:/ruby/lib/ruby/1.8/tk.rb:1860:in `__invoke'
    c:/ruby/lib/ruby/1.8/tk.rb:1860:in `_invoke'
    ...
    c:/ruby/lib/ruby/1.8/tk/canvastag.rb:48:in `configure'
```

```
    svgcanvas.rb:123:in `select'
    svgcanvas.rb:192:in `putSvgIntoCanvas'
...
    svgcanvas.rb:538:in `test_tab'
```

Something called `.w00007` broke, deep in Tk. The call stack at line `123`, the lowest stack frame in our program, indicates the statement `$lastItem.configure('outline', 'black')`. As predicted, our `$lastItem` bubble has burst. It still points to the last item selected in the previous test. That item belongs to the previous test's canvas, so Ruby throws an error.

Note that, despite TFP's ability to reconstruct many other best design practices—such as abhoring global variables—my unearthly laziness also played a role.

> I delayed creating an otherwise very obvious class until a failing test forced me to.

We need to suspend the current test, then refactor to wrap the TkCanvas handle up in a new class, because we finally found a member variable for it. The Simplicity Principles don't forbid writing a speculative class, even one without variables. I don't know if writing one before now would have slowed me down or sped me up. I am now certain that I know more about how to create this class than I did before.

As Vera Peeters says: when you are afraid of something, do it more often.

However, the features that provided evidence we need a class are also the ones that get in the way of refactoring it. Disable these new tests:

```
    def pest_selectNode()
...
    end

    def pest_tab
...
    end
```

Disable the new function that crashes:

```
def select(item)
    return
...
end
```

The tests all pass again.

When suspending a new feature to refactor, you might erase your source and return to the latest integration, depending on how invasively the new feature changed things, and how long you expect the refactor to take.

Extract Class Refactor

The `*Builder` classes each will need to take a reference to our new object, when it exists, so they can each add their item to its canvas. Another good thing about Ruby is we can use objects that pretend to be instances of a class that does not exist yet.

Find each version of `newThing()`, and add this new object to their signatures:

```
    def newThing(anSvgCanvas, canvas, svg)
…
    end
```

Don't do anything with it yet. The second canvas parameter is now deprecated.
Find the single place where the system calls `newThing()`, and add a `nil` placeholder:

```
      aBuilder.newThing(nil, canvas, svgThing)
```

Changing method signatures is always the riskiest part of refactoring, so do it first. Now that all the tests pass, we are free to go to the next step.

Create a new class, copy `select()` into it, change that naughty global `$lastItem` into a `@lastItem`, making it a member:

```
class SvgCanvas
    def select(item)
        if @lastItem then
            @lastItem.configure('outline', 'black')
            @lastItem.configure('width', 1.0)
        end

        item.configure('outline', 'green')
        item.configure('width', 3.0)
        @lastItem = item
    end
end
```

Then instantiate that class, pass it into `newThing()`, and use it for the `'Key-Tab'` event handler:

```
def putSvgIntoCanvas(svg, canvas)

    anSvgCanvas = SvgCanvas.new()

    doc = Document.new(svg)

    builders = [
        EllipseBuilder.new(),
        PolygonBuilder.new(),  #  behind the TkcText items
        TextBuilder.new(),
        PathBuilder.new(),
        ]

    builders.each() do |aBuilder|
        XPath.each(doc, '/svg/g/g/' + aBuilder.svgTag()) do
            |svgThing|
            aBuilder.newThing(anSvgCanvas, canvas, svgThing)
        end
    end

    canvas.bind('Key-Tab') {
        if canvas.find_all.size() > 0 then
            anSvgCanvas.select(canvas.find_all[0])
        end
        }
    canvas.focus()  #  so keystrokes don't go into the frame
    return anSvgCanvas  #  a test might need it
end
```

The tests all pass. That only proves our new class compiles and instantiates correctly. The new `.select()` method could have many errors, because dynamically typed languages like Ruby defer much type checking until runtime.

Next, erase the old `select()` function, and put the new new `.select()` method online:

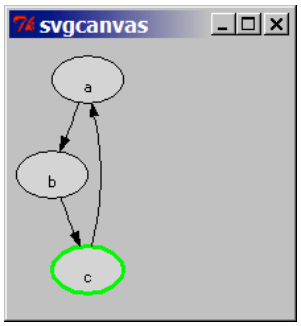
```
    class EllipseBuilder < Builder
…
        def newThing(anSvgCanvas, canvas, svg)
            element = constructor(canvas, svg, TkcOval)
            style = parseStyle(svg.attributes['style'])
            configureStyle(element, style, 'outline', 'stroke')
            configureStyle(element, style, 'fill', 'fill')

            element.bind('Button-1') {
                anSvgCanvas.select(element)
                }
        end
    end
```

Nothing calls it yet, because its tests are commented out. The other tests, of course, still pass. Activate the tests that call the new `.select()`:

```
        def test_selectNode()
    …
        end

        def test_tab
    …
        end
```

And the tests pass. We will confirm things with a *Temporary Interactive Test*:

```
        def sampleDOT()
            return 'digraph aGraph {
                        node [style=filled];
                        a -> b -> c -> a }'
        end

        def test_tab()
            svg = generateSvg(sampleDOT())

            putSvgIntoCanvas(svg, @canvas)
            assert(!isSelected(0))
            click(@canvas, 'Key-Tab')
            assert(isSelected(0))
            maybeMainloop(true)
        end
```



Clicking on any node "moves" the selection emphasis to it. (We really know the thick green oval doesn't really *move*. One node turns the emphasis off and another turns it on. Users perceive this as motion.)

However, <Tab> always moves the emphasis to the "a" node. We will review our architecture, then fix that.

Ideally, many methods that take a `canvas` argument could take `anSvgCanvas`.. That class is now properly instantiated, so future refactors can migrate features into it. However, nobody knows about the `anSvgCanvas` except two block closures that handle events:

```
            element.bind('Button-1') {
                anSvgCanvas.select(element)
                }
...
        canvas.bind('Key-Tab') {
            if canvas.find_all.size() > 0 then
                anSvgCanvas.select(canvas.find_all[0])
            end
            }
```

Maybe, some day, more things that use `canvas` will migrate into `anSvgCanvas`. Presently, our only reason to put everything into that class would be the *Clear and Expressive* Simplicity Principle. That would help production code, but not a book, so we will just explain why this funny business works, then fix the <Tab> issue.

In a language without block closures, `anSvgCanvas` must maintain a scope and lifespan long enough to serve one test case (or one `main()` function). Ruby objects, by contrast, live as long as something references them. Here, `canvas` refers to a `TkCanvas` that refers to `TkcOval` objects, which refer to bound events referring to `Proc` objects referring to block closures referring to `anSvgCanvas`. So `anSvgCanvas` will live as long as each test case's `@canvas` object. If this curious situation ever becomes a problem, a fix is a refactor away.

Keyboard Navigation

The <Tab> key goes to the "a" node because this statement told it to:

```
            anSvgCanvas.select(canvas.find_all[0])
```

To pass `test_tab()`, we merely jump to the first canvas item.

Now write a test to force a second <Tab> keystroke to select the second item:

```
    def test_secondTab()
        svg = generateSvg(sampleDOT())
```

```
        SvgCanvas.new(@canvas).putSvgIn(svg)
        assert(!isSelected(0))
        click(@canvas, 'Key-Tab')
        assert(isSelected(0))

        click(@canvas, 'Key-Tab')
        assert(isSelected(1))
        maybeMainloop(true)
    end
```

The `maybeMainloop(true)` will Temporarily Interactively Test that `click(@canvas, 'Key-Tab')` does the same thing as a real <Tab> key does.

The test doesn't pass yet; the code reading the SVG currently throws away information we need.

Child Test

We must temporarily disable the test, and write another lower-level test that prevents our canvas system from throwing this information away. The test uses our trivial loop again, but checks we stored the name that `dot` gave our node in its tag:

```
    def pest_secondTab()
…
    end

    def test_setTags()
        svg = generateSvg(sampleDOT())

        putSvgIntoCanvas(svg, @canvas)
        oval = @canvas.find_all[0]
        assert_equal('node1', oval.gettags()[0])
        maybeMainloop()
    end
```

Each canvas item has an array of tags containing application specific strings. Canvases can use tags to query out subsets of their item set, so these tags are about to become very useful.

But to pass that test, we must un-throw away information found in the intermediate SVG. I marked the string we need to store in that tag in **`bold`**:

```
<g id="node1" class="node"><title>a</title>
<ellipse cx="65" cy="29" rx="54" ry="18"
      style="fill:none;stroke:black"/>
<text text-anchor="middle" x="65" y="34">Harry Potter</text>
</g>
```

`putSvgIntoCanvas()` queries past the `<g>` node and fetches the `<ellipse>` node with XPath. The query resolves to `/svg/g/g/ellipse`:

```
    builders.each() do |aBuilder|
        XPath.each(doc, '/svg/g/g/' + aBuilder.svgTag()) do
            |svgThing|
            aBuilder.newThing(anSvgCanvas, canvas, svgThing)
        end
    end
```

`newThing()` will need to know the `<g>` tag contents wrapping the `<ellipse>` object. It receives `svgThing`, containing the `<ellipse>` object, but not `doc` or the XPath address. To "reach backwards", and get a containing item's attributes, we can use REXML's Document Object Model, by querying the `@id` from the `ellipse`'s `parent`:

```
    class EllipseBuilder < Builder
...
        def newThing(anSvgCanvas, canvas, svg)
            element = constructor(canvas, svg, TkcOval)
            style = parseStyle(svg.attributes['style'])
            configureStyle(element, style, 'outline', 'stroke')
            configureStyle(element, style, 'fill', 'fill')
            element.addtag(svg.parent.attributes['id'])

            element.bind('Button-1') {
                anSvgCanvas.select(element)
                }
        end
    end
```

We will finish the <Tab> feature soon. The `id`s, stored in the `tag` configurations, will tell the <Tab> event handler how to move the selection emphasis to the next node. First, a reminder where this project has been, where it's going, and how a bug in our Representation Layer will cause the next problem.

Total Recall

To draw and edit a family tree, we use GraphViz's DOT notation as the Logic Layer, and SVG as our Representation Layer. The `dot` program stuffs SVG with more information than the last chapter needed; it merely painted SVG graphics. That chapter also squeezed down code in a way that discarded access to that information. The method that extracts a graphical item the SVG moved far away from the one that draws it on the canvas.

This Case Study converts the read-only canvas to a read-write one. That requires transferring more information from the SVG into each canvas item. A real project would have grown the output features together with the input ones. Developing features out of order illustrates how Test-First Programming can deal with the dreaded "changing requirements" problem.

Interactivity requires the concept of a "selected item", so our first new feature highlights clicked-on items with a thick green border. Green means "Go". But to let <Tab> cycle through the items, the statements we will bind to the `'Key-Tab'` event must be able to extract from a sea of items the "next" one.

GraphViz's dot gave us this information, in parts of the SVG we threw away. That system gave shapes corresponding to graph nodes an `id="node1"`. To find the "next" node we need to find the next `id` in the SVG. This simple feature becomes foundational for further features that will use more of the information dot wrote into the SVG.

However, GraphViz's `dot` command has a bug, so we will temporarily instrument a test to reveal it:

```
        def test_setTags()
            svg = generateSvg(sampleDOT())

            putSvgIntoCanvas(svg, @canvas)
            oval = @canvas.find_all[0]
```

```
        assert_equal('node1', oval.gettags()[0])

        puts(svg)

        maybeMainloop()
    end
```

Now install the mighty CygWin platform, to get `grep`, and enter this command line:

```
> ruby svgcanvas.rb | grep node
<g id="node1" class="node"><title>a</title>
<g id="node3" class="node"><title>b</title>
<g id="node4" class="node"><title>c</title>
```

The bug is "`node2`" is missing. This makes the nearly monotonic numbering of the other `id`s useless. Without the bug, to tab from `node1` to `node2`, we would extract the `1`, increment it, convert it to `"node2"`, and fetch. With this bug, to tab from `node1` to `node3`, we must store a table of these nodes, and

Fixing this bug, down in `dot`'s source, is outside this book's scope. We must treat `dot` as black-box, and write a test that invests a fix into the `SvgCanvas` class:

```
    def test_storeTags()
        svg = generateSvg(sampleDOT())

        anSvgCanvas = putSvgIntoCanvas(svg, @canvas)

    #   warning - this test preserves an insect in GraphViz. When it
    #   upgrades, the insect might go away or change
    #   The bug is the nodes skip 2: "node1", "node3", "node4"

        assert_equal("node3", anSvgCanvas.getNext("node1"))
        assert_equal("node4", anSvgCanvas.getNext("node3"))
        assert_equal("node1", anSvgCanvas.getNext("node4"))
        assert_equal("node1", anSvgCanvas.getNext("spinach"))

        maybeMainloop()
    end
```

This test should fail if `dot` upgrades and changes the bug, so we can reevaluate our options. Always fix a bug as close as possible to its source.

Run that test, and predict a syntax error. `anSvgCanvas.getNext()` does not exist yet. Add it…

```
    class SvgCanvas

        def getNext(nodeName)

        end
…
    end
```

…and run the tests again, this time predicting failure. Now pass the test:

```
    class EllipseBuilder < Builder
…
```

```
        def newThing(anSvgCanvas, canvas, svg)
…
            nodeName = svg.parent.attributes['id']
            element.addtag(nodeName)
            anSvgCanvas.addNodeName(nodeName)
    …
        end
    end

    class SvgCanvas

        def addNodeName(nodeName)
            @nodeNames = []  if not @nodeNames
            @nodeNames << nodeName
        end

        def getNext(nodeName)
            index = @nodeNames.index(nodeName)
            return @nodeNames[0]  if not index
            newNode = @nodeNames[index + 1]
            return (if newNode then newNode else @nodeNames[0] end)
        end
…
    end
```

The tests all pass, and Ruby's variable syntax needs more discussion. The first use of a variable name returns the `nil` object. `if not` evaluates `true` when its So this line…

```
            @nodeNames = []  if not @nodeNames
```

…initializes the member `@nodeNames` to an array, if it did not already exist.

The `<<` operator appends an element to an array.

The `.index()` method returns the index of an array element, or `nil` if it is "`spinach`".

Indexing an array to fetch a value, `@nodeNames[index + 1]`, returns `nil` if the index is out of bounds. So `.getNext()` returns the first node's name if we pass it an invalid node name, or if the user tabs off the end of the node list.

`if 0` evaluates as `true`. Conditionals refuse `nil` or `false`, so `0` is just another number. And `if` yields the value of its successful controlled block.

Ruby's careful balance of features, including permitting variables and array elements that don't exist to evaluate to `nil`, permits statements that work correctly by default. A harder language would require extra `if` conditions, leading to more source statements. Put another way, Ruby itself took care of duplicated definitions of behavior for us.

Tab to the Next Node

Restore this test:

```
        def test_secondTab()
            svg = generateSvg(sampleDOT())

            putSvgIntoCanvas(svg, @canvas)
            assert(!isSelected(0))
            click(@canvas, 'Key-Tab')
            assert(isSelected(0))

            click(@canvas, 'Key-Tab')
```

```
            assert(isSelected(1))
            maybeMainloop(true)
        end
```

Run the tests, and successfully predict the same failure as before, at `assert(isSelected(1))`. This reaffirms our edits didn't unexpectedly change the infrastructure.

Now pass the test:

```
    class SvgCanvas
...
        def selectNext(canvas)
            nodeName = @lastItem.gettags[0]  if @lastItem
            nextNodeName = getNext(nodeName)
            select(canvas.find_withtag(nextNodeName)[0])
        end
    end

        canvas.bind('Key-Tab') {
            if canvas.find_all().size() > 0 then
                anSvgCanvas.selectNext(canvas)
            end
            }
```

The test temporarily raises our three familiar nodes, and <Tab> cycles between them.

This system coupled the <Tab> key to the order of nodes declared in an input DOT file. If our input files get bigger, the <Tab> key's behavior will become counterintuitive. Future work in this direction should provide support for the arrow keys, by selecting the closest node within a quadrant.

Edit a Node

To change the name of a family member in our tree, we want an edit field, floating inside the canvas, over the node it will rename. Think of how your file explorer renames:

To learn how to get that effect, write a Temporary Interactive Learner Test to see how TkCanvas permits inserting other Tk controls as canvas items:

```
        def test_addEditField()
            edit = TkText.new()
            TkcWindow.new(@canvas,0,0,'anchor'=>'nw','window'=>edit)
            maybeMainloop(true)
        end
```

Running the tests now raises a canvas with an edit field in its upper left corner. I wrote on the edit field, to confirm it behaves like an edit field. Then I took the `true` out of `maybeMainloop()`.

The next test that makes <F2> switch the currently selected node into edit mode. Notice the `click()` function, which so diligently conveyed mouse clicks to nodes a few pages back, now just as easily conveys keystrokes to windows. This emergent success combined luck, Tk's coherent architecture, and a little planning. I gave `click()` the right arguments to enable these abilities:

```
        def test_f2_editsNode()
            svg = generateSvg(sampleDOT())

            putSvgIntoCanvas(svg, @canvas)
            oval = @canvas.find_all[0]
            max = @canvas.find_all.length()
            click(oval, 'Button-1')         #  click the "a" node
            click(@canvas, 'Key-F2')        #  switch it to edit mode
            window = @canvas.find_all[max]  #  find the editor

            assert_kind_of(TkcWindow, window)
            editField = window.cget('window')
            assert_kind_of(TkText, editField)

          #  and verify it edits the "a" node

          #  assert_equal('a', editField.value)
            maybeMainloop()
        end
```

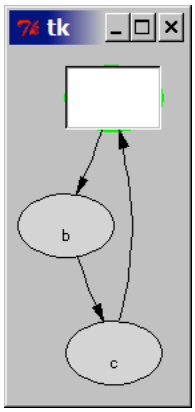
That test forces these edits:

```
    class SvgCanvas
…
        def editCurrentNode(canvas)
            if @lastItem then
                edit = TkText.new()
                x1,y1,x2,y2 = @lastItem.bbox()

                TkcWindow.new(canvas, x1, y1,
                                      'window' => edit) do
                    width   x2-x1
                    height  y2-y1
                    anchor 'nw'
                end
            end
        end

    end

    def putSvgIntoCanvas(svg, canvas)
…
        canvas.bind('Key-F2') {
            anSvgCanvas.editCurrentNode(canvas)
            }
…
```

```
        end
```

This keystroke handler binds to the canvas because its items cannot bind keystrokes.

We create a `TkcWindow`, bond it with a `TkText` field, and position it directly on top of the graph node. I "authored" code to position the edit field on top of the target node—tests don't strictly constrain the edit field's position, but they should be easy to add. New, similar tests should ensure the `TkcWindow` goes away when dismissed; that <Enter> saves changes and <Escape> discards them, etc. This Case Study focuses on hard GUI features that require architectural changes. Our `maybeMainloop(true)` reveals this:



The edit field lacks an "a". It must contain the existing label, for editing. That's why the last test case commented-out its last assertion. Getting that text requires a little more infrastructure.

At <F2> time, `anSvgCanvas` knows the selected node, but not its text. The user sees these together, because dot wrote them like that, but the canvas only sees a list of graphical elements. To find that "a" and write it in the edit field, we could use `.find_above()` to reach from the node element to its text element, but that would introduce risk and fragility. `dot` can be forced to push those elements apart from each other.

Canvas items store data in an array of tags. The `EllipseBuilder` will query its `<ellipse>` node's peer `<text>` node, and write its contents into the new `TkcOval`'s second tag element. At <F2> time, a canvas item will present the text found in its own second tag. So once again, we leave a test unfinished while writing a Child Test to add infrastructure:

```
        def test_storeTextInTags()
            svg = generateSvg(sampleDOT())
            putSvgIntoCanvas(svg, @canvas)
            node1 = @canvas.find_all[0]
            node2 = @canvas.find_all[1]
            node3 = @canvas.find_all[2]

            assert_equal("a", node1.gettags[1])
            assert_equal("b", node2.gettags[1])
            assert_equal("c", node3.gettags[1])

            maybeMainloop()
        end
...
    class EllipseBuilder < Builder
...
```

```
        def newThing(anSvgCanvas, canvas, svg)
            element = constructor(canvas, svg, TkcOval)
            nodeName = svg.parent.attributes['id']
            element.addtag(nodeName)
            peerNode = svg.parent.get_elements('text')[0]
            nodeText = peerNode.text()  if peerNode
            element.addtag([nodeText])  if nodeText
…
        end
    end
```

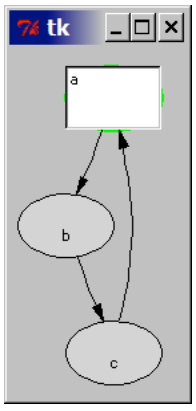
That all works. Now de-comment this line from `test_f2_editsNode()`:

```
            assert_equal('a', editField.value)
```

…and get it to pass:

```
        def editCurrentNode(canvas)
            if @lastItem then
                edit = TkText.new()
                x1,y1,x2,y2 = @lastItem.bbox()
                nodeText = @lastItem.gettags()[1]
                edit.value = nodeText  if nodeText

                TkcWindow.new(
                    canvas, x1, y1,  #  an alternative constructor style
                    :window => edit,
                    :width  => x2-x1,
                    :height => y2-y1,
                    :anchor => 'nw'
                    )
            end
        end
```

Voila:



You and I have both forgotten when I stopped reminding you to switch between `maybeMainloop()` and `maybeMainloop(true)` to temporarily review test results. Freely switching between silent and visible tests drives TFUI's *Flow* Principle.

Similarly, our habit of writing a GUI test, suspending it, adding new infrastructure, and finishing that test will continue.

Write DOT files

To edit node labels, and save new DOT files with new contents, first we must change the text in the canvas. Before that, we need to write half a *Loose User Simulation* test to change an edit field's context, and temporarily turn on the Event Queue spigot to visually confirm the simulation worked before finishing the other half of the test. (Pretend you did not read Part I, and read that sentence again!)

This *Temporary Visual Inspection* ensures that our tests can change that little edit field's text:

```
def test_enter_commitsNode()
    svg = generateSvg(sampleDOT())
    putSvgIntoCanvas(svg, @canvas)

    oval = @canvas.find_all()[0]
    max = @canvas.find_all().length()
    click(oval, 'Button-1')
    click(@canvas, 'Key-F2')
    window = @canvas.find_all()[max]
    editField = window.cget('window')
    editField.value = 'Grandma'
    maybeMainloop(true)
end
```

That displays this…

…and we are not ready to finish the test. If we did, the case would send a `'Key-Return'` event to the `editField`, then it would assert that the associated `TkcText` item has the new text.

However, "`a`" is smaller than "`Grandma`". Correctly refreshing the `TkcText` field's size requires a round trip through `dot`. And that requires a whole new facility. Our canvas needs the ability to read each item and write a line to a new DOT script, defining that item.

DOT makes this easy.

MetaData

From the last Case Study until here, this project has read SVG and written a canvas. To close the loop, and permit canvas edits, we must read a canvas and write DOT notation. Think about the problems that causes.

Suppose a canvas contains a `TkcLine`, and we expect to reconstitute DOT notation from it, such as "`a -> b`". The canvas contains no awareness that this `TkcLine` has any special relationship with any `TkcOval` at one end or another.

However, `dot` writes extra SVG data, in non-graphic `<title>` tags, to cover this situation. To close our loop, we must keep that data in the loop.

Here's a dirt-simple DOT file:

```
digraph aGraph { a -> b }
```

And here's a detail of its SVG output:

```
<g id="edge2" class="edge">
  <title>a-&gt;b</title>
```

```
  <path style="fill:none;stroke:black;"
        d="M42,63C42,74 42,86 42,98" />
  <polygon style="fill:black;stroke:black;"
           points="46,98 42,111 39,98 46,98" />
</g>
```

I marked the `<title>` tags we need in **bold**. "`a-&gt;b`" is XML's way of saying "`a->b`". That `->` arrow has been right under our nose, all along.

If we use `->` to mean "begat", a family tree would look like this:

```
digraph aGraph {
    person1 [label="Nana Oyl"];
    person2 [label="Castor Oyl", shape=box];
    person1 -> person2; }
```

That DOT notation uses `shape=box` for males (at time of birth). And it doesn't use the node labels as node names; that would cause trouble as the node contents get richer. The notation uses boring monotonic node names of `person1`, `person2`, etc. Personal names can contain more kinds of characters than DOT identifiers.

The loop these data travel around is DOT → SVG → TkCanvas → new DOT files. We have studied the data model enough to develop a strategy. We will stuff more metadata into some canvas items' `tags` configurations. Each tag is an array element, and the italic entries in this table are finished:

class	gettags[0]	gettags[1]	gettags[2]
node	*id="**node1**"*	*<text>**label**</text>*	<title>**person1**</title>
edge	id="**edge1**"	nil	<title>**person1->person2**</title>

When building a canvas, `TkcOval` objects (and eventually some `TkcPolygon` objects) will Set their DOT node names in their `gettags[2]` configuration. `TkcLine` objects will Set their DOT edge commands into their `gettags[2]` configurations. When converting a canvas to a DOT script, we need only extract all these tags and add them up.

There are other ways to do this. We could invent a new file format. We could read the input file in DOT notation, parse it ourselves, maintain a data model in memory, change this at edit time, and traverse the data model to write a new file. But those options would duplicate abilities, including the ability to parse DOT notation. Sometimes projects can use the Duplicate Observed Data Refactor to separate a data model in logic from its copy in a GUI.

With our planned system, when the user changes a family member's name, we write the new name into their nodes' `gettags[2]` slot. Then we generate a new DOT script, erase all canvas items, call `putSvgIntoCanvas()` again to refresh all the items, and re-select the current node. Sometimes stuffing data into a GUI control is not good. In this case it's perfectly reasonable. The authors of GraphViz and Tk provided these abilities, so we only need reuse them.

Our next test forces our canvas to follow that schedule:

```
def generateNanaOyl()
    svg = generateSvg('digraph aGraph {
        node [style=filled];
        person1 [label="Nana Oyl"];
        person2 [label="Castor Oyl", shape=box];
        person1 -> person2; }')

    aCanvas = SvgCanvas.new(@canvas)
```

```
            aCanvas.putSvgIn(svg)
            return aCanvas
        end

        def test_storeTitleInSecondTag()
            generateNanaOyl()
            oval, box, arrow, text1, text2, line = @canvas.find_all
            assert_kind_of(TkcOval,    oval )
            assert_kind_of(TkcPolygon, box  )
            assert_kind_of(TkcPolygon, arrow)
            assert_kind_of(TkcText,    text1)
            assert_kind_of(TkcText,    text2)
            assert_kind_of(TkcLine,    line )

     assert_equal( [ 'node1',   'Nana Oyl', 'person1' ], oval .gettags() )
     assert_equal( [ 'node2', 'Castor Oyl', 'person2' ], box  .gettags() )
     assert_equal( [                                   ], arrow.gettags() )
     assert_equal( [                                   ], text1.gettags() )
     assert_equal( [                                   ], text2.gettags() )
     assert_equal( [ 'edge2', '',  'person1->person2' ], line .gettags() )

            maybeMainloop()
        end
```

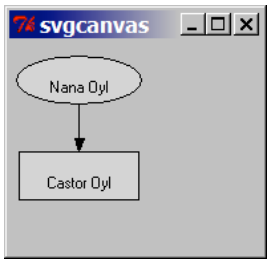
Those table's columns look suspiciously similar to our design plan.

After writing each failing `assert_equal()` in that test, I made the source pass it before adding the next one. The upgrades and (mild) refactors accumulate into:

```
    class PathBuilder < Builder
...
        def newThing(anSvgCanvas, canvas, svg)
            style = parseStyle(svg.attributes['style'])

            if 'none' == style['fill'] then
                element = constructor(canvas, svg, TkcLine)
                configureStyle(element, style, 'fill', 'color')
            else
                element = constructor(canvas, svg, TkcPolygon)
                configureStyle(element, style, 'outline', 'stroke')
            end

            element.configure('smooth', '1')
            nodeClass = svg.parent.attributes['class']

            if 'edge' == nodeClass then  #  only register edges
                anSvgCanvas.registerNode(element, svg)
            end

        end
    end

    class EllipseBuilder < Builder
...
        def newThing(anSvgCanvas, canvas, svg)
            element = constructor(canvas, svg, TkcOval)
            style = parseStyle(svg.attributes['style'])
            configureStyle(element, style, 'outline', 'stroke')
            configureStyle(element, style, 'fill', 'fill')

            nodeName = anSvgCanvas.registerNode(element, svg)
```

```
            anSvgCanvas.addNodeName(nodeName)

            element.bind('Button-1') {
                anSvgCanvas.select(element)
                }
        end
    end

    class PolygonBuilder < Builder
...
        def newThing(anSvgCanvas, canvas, svg)
            element = constructor(canvas, svg, TkcPolygon)
            style = parseStyle(svg.attributes['style'])

            nodeClass = svg.parent.attributes['class']

            if 'edge' != nodeClass then  #  don't register arrowheads
                nodeName = anSvgCanvas.registerNode(element, svg)
                anSvgCanvas.addNodeName(nodeName)

                element.bind('Button-1') {
                    anSvgCanvas.select(element)
                    }
            end

            configureStyle(element, style, 'outline', 'stroke')
            configureStyle(element, style, 'fill', 'fill')
            element.configure('width', 1)
        end
    end

    class SvgCanvas
...
        def registerNode(element, svg)
            nodeName = svg.parent.attributes['id']
            element.addtag(nodeName)
            peerNode = svg.parent.get_elements('text')[0]
            nodeText = peerNode.text()  if peerNode
            element.addtag([nodeText])
            peerTitle = svg.parent.get_elements('title')[0]
            titleText = peerTitle.text()  if peerTitle
            element.addtag([titleText])
            return nodeName
        end

    end
```

Further refactors should find ways to merge those excess `if` statements. Special cases happen when one class does two or more things. Here, `PolygonBuilder`, for example, constructs both arrowheads and boxes:



Only boxes should respond to the user. A better system could introduce into two derived Builder types, with methods. The Replace Conditional with Polymorphism Refactor is the heart of Object Oriented Programming.

Going in smaller steps, the new code at least takes care of an item on our Goal Stack. Until now, boxes did not respond to the user at all. I authored those features while merging the common code that Sets `tags`.

Structured Programming

Before Object Oriented Programming, people who studied the difference between cruft and maintainable code learned to improve things by linking control flow to variable scope. An example in C:

```
if (whatever())
{
    int x = something();
    somethingElse(x);
}
x;  /*  this line won't compile, unless another x is available from above  */
```

Structured Programmers decry the mighty `goto` keyword, able to leap tall procedures in a single bound. When a language permits a `goto` unfettered by that language's block system, `{}`, variables defined in statements controlled by `goto` have ambiguous scope. Structured Programming enables the style guideline "give identifiers the most restricted scope possible".

I am unaware if Ruby supports `goto`. Object Oriented Programming subsumes all Structured Programming ideals. However, Ruby permits this:

```
def newThing()

    if something() then
        element = constructor(TkcLine)
    else
        element = constructor(TkcPolygon)
    end

    element.configure('smooth', '1')

end
```

Ruby creates variables by assigning to them. That cures an ancient problem with the C languages, where variables can define without values. The lowly C statement `int x;` efficiently produces an `x` that might contain anything, and should not be used until assigned. Ruby fixes that problem by forbidding creation without assignment.

Ruby variables create into their method's scope, not their block's scope. That's why the above code works—it tweaks the Structured Programming rules. The `element` inside the `if` statement's blocks should, in theory, exit scope and disappear before the last `element`.

Ruby supports Structure Programming much more subtly than by decrying `goto`. When a `goto` leaps a tall procedure in a single bound, the root problem is that tall procedure itself. Ruby's syntax maximizes opportunities to minimize code. The style guideline "don't `goto`" is subservient to the Simplicity Principle *Minimize Methods*.

For example:

```
        def newThing()
            klass = if something() then TkcLine else TkcPolygon end
            element = constructor( klass )
            element.configure('smooth', '1')
        end
```

That example returns a class, as an object, from an `if` statement, then passes it into a method. That example illustrates why the previous Case Study did not begin by diving into Ruby's deep end.

Convert a Canvas to DOT Notation

Stashing that much information into the canvas tags might let us reconstitute DOT files from them. Per You Aren't Gonna Need It, again, we don't need a system that can reconstruct any possible DOT file from any possible canvas. Here, our DOT files only use a few of their features—enough to draw family trees—so the canvas only needs to write those few features into new DOT files.

To write a DOT file, first we must write DOT lines. So we need a function that turns one canvas item into a string. The string may contain zero or more lines of DOT commands.

First, the test covers all the situations where the canvas item must not return anything:

```
        def test_writeDOTstatement()
            aCanvas = generateNanaOyl()
            oval, box, arrow, text1, text2, line = @canvas.find_all

            assert_equal('', aCanvas.writeDOTstatement(text1))
            assert_equal('', aCanvas.writeDOTstatement(text2))
            assert_equal('', aCanvas.writeDOTstatement(arrow))

        end
...
    class SvgCanvas
...
        def writeDOTstatement(element)
            return ''
        end
    end
```

That was easy! Now create a line of DOT from an `oval`:

```
            assert_equal( "person1 [label=\"Nana Oyl\"];\n",
                           aCanvas.writeDOTstatement(oval) )
...
        def writeDOTstatement(element)
            tags = element.gettags()
            return ''  if tags == []
            nodeName  = tags[2]
            nodeLabel = tags[1]
            box = ''
            box = ', shape=box'  if element.instance_of?(TkcPolygon)
            return "#{nodeName} [label=\"#{nodeLabel}\"#{box}];\n"
        end
```

That maze of escape codes passes our test.

Now add more assertions, testing after each one, and predict `box` will pass and `line` will fail:

```
        assert_equal( "person1 [label=\"Nana Oyl\"];\n",
                      aCanvas.writeDOTstatement(oval) )

        assert_equal( "person2 [label=\"Castor Oyl\", shape=box];\n",
                      aCanvas.writeDOTstatement(box) )

        assert_equal('', aCanvas.writeDOTstatement(text1))
        assert_equal('', aCanvas.writeDOTstatement(text2))
        assert_equal('', aCanvas.writeDOTstatement(arrow))

        assert_equal( "person1->person2;\n",
                      aCanvas.writeDOTstatement(line) )
```

The last assertion fails with:

```
test_writeDOTstatement(TestGrapher) [svgcanvas.rb:760]:
<"person1->person2;\n"> expected but was
<"person1->person2 [label=\"\"];\n">.
```

That diagnostic reveals how to upgrade the production code:

```
    def writeDOTstatement(element)
        tags = element.gettags()
        return ''  if tags == []
        nodeName  = tags[2]
        nodeLabel = tags[1]
        return "#{nodeName};\n"  if nodeLabel == ''
        box = ''
        box = ', shape=box'  if element.instance_of?(TkcPolygon)
        return "#{nodeName} [label=\"#{nodeLabel}\"];\n"
    end
```

`writeDOTstatement()` now handles every item type, so use it to write a DOT file. We shouldn't write a test like this:

```
        def test_writeDOTfile()
            aCanvas = generateNanaOyl()
            dot = aCanvas.writeDOTfile(@canvas)

            assert_equal("digraph aGraph {\n" +
                "node [style=filled];\n" +
                "person1 [label=\"Nana Oyl\"];\n" +
                "person2 [label=\"Castor Oyl\", shape=box];\n" +
                "person1 -> person2;\n" +
                "}", dot)
        end
```

That violates the principle "don't compare very long strings". The *Fuzzy Matches* Principle (on page 37) addresses environments that force that principle to bend. However, whenever we add more features to this Family Tree project, that DOT string will change, so fuzziness defeats this test's purpose.

The best strategy here lets `dot` test for us—a Parsed Fuzzy Match:

```
    def test_writeDOTfile()
```

```
        aCanvas = generateNanaOyl()
        dot = aCanvas.writeDOTfile(@canvas)

        f = open('family.dot', 'w')
        f.write(dot)
        f.close()

        result = `dot -Tplain family.dot 2>&1`
        puts(result)
    end
```

Ruby evaluates strings inside back-ticks, ``, by executing a command line and piping its output as a string. The command line's `2>&1` notation redirects `STDERR` into `STDOUT` so it can appear in `result`. Many command line environments support that notation, but it's not on Ruby's side.

To help this test learn to detect errors in DOT files, we temporarily put an error into a DOT file:

```
    class SvgCanvas
…
        def writeDOTfile(canvas)
            return 'an error'
        end
    end
```

When `dot` encounters a syntax error, it writes a complaint to the `STDERR` channel containing, among other things, "`>>>`":

```
Error: family.dot:0: parse error near line 0
context:  >>> an <<<  error
```

That tells us how to make the test fail for the correct reason:

```
        def test_writeDOTfile()
            aCanvas = generateNanaOyl()
            dot = aCanvas.writeDOTfile(@canvas)

            f = open('family.dot', 'w')
            f.write(dot)
            f.close()

            result = `dot -Tplain family.dot 2>&1`
            assert_no_match(/\<\<\</, result)
        end
```

That assertion will fail for any non-trivial contents of "`family.dot`" which the `dot` utility does not like. It won't test that "`family.dot`" contains a valid family tree, but it will at least fail for many kinds of errors we could introduce while generating DOT notation.

After we fix `aCanvas.writeDOTfile()`, `result` will contain this (`shrunk to fit`):

```
graph 1.000 1.250 1.722
node person1  0.625 1.361 1.254 0.500 "Nana Oyl" filled ellipse black lightgrey
node person2  0.625 0.361 1.222 0.500 "Castor Oyl" filled box black lightgrey
edge person1 person2 4 0.625 1.111 0.625 1.000 0.625 0.875 0.625 0.750 solid black
stop
```

The command `dot -Tplain` produces a nice report, in inches, of where `dot` was going to place the nodes and edges. Tests could parse that, if we needed more aggressive checks on its contents. (Or we could use `dot -Tsvg`, and start the data lifecycle over again…)

This code makes `dot` happy:

```
def writeDOTfile(canvas)
    dot = "digraph aGraph {\n" +
        "node [style=filled];\n"

    canvas.find_all.each() do |item|
        dot += writeDOTstatement(item)
        end

    dot += "}"
    return dot
end
```

Back on page 138, we left a test unfinished. Don't thumb back—I'l just copy it here:

```
def test_enter_commitsNode()
    svg = generateSvg(sampleDOT())
    putSvgIntoCanvas(svg, @canvas)

    oval = @canvas.find_all[0]
    max  = @canvas.find_all.length()
    click(oval, 'Button-1')
    click(@canvas, 'Key-F2')
    window = @canvas.find_all()[max]
    editField = window.cget('window')
    editField.value = 'Grandma'
    maybeMainloop()
end
```

That displays this…

…and we might be ready to finish the test. It will send a `'Key-Return'` event to the `editField`, then assert that the associated `TkcText` item has the new text:

```
def test_enter_commitsNode()
    generateNanaOyl()  #  switch to Family Tree Format

    oval = @canvas.find_all[0]
    max  = @canvas.find_all.length()

    click(oval, 'Button-1')
    click(@canvas, 'Key-F2')

    window    = @canvas.find_all()[max]
    editField = window.cget('window')
    editField.value = 'Grandma'

    click(editField, 'Key-Return')

    # p(@canvas.find_all.collect{|n| n.class.name })
```

```
        text = @canvas.find_all[3]
        assert_equal('Grandma', text.text)
        assert(isSelected(0))

        maybeMainloop()
    end
```

That innocent-looking test belies the production code's awesome responsibilities at <Enter> time. The code will Set "Grandma" into her node's tags, use the tags to write a new DOT script, convert it to SVG, erase all the canvas controls (including that edit field), put the SVG into the canvas, and reselect Grandma's node:

```
    class SvgCanvas
...
        def canvasToSvg(canvas)
            dot = writeDOTfile(canvas)
            open('scratch.dot', 'w') do |f| f.write(dot) end
            system 'dot -Tsvg scratch.dot -o graph.svg'
            open('graph.svg', 'r') do |f| return f.read()  end
        end

        def upgradeCanvas(canvas)
            svg = canvasToSvg(canvas)

          #  preserve current node selection, then croak all items

            currentNode = @lastItem.gettags[2]  if @lastItem
            canvas.remove('all')

          #  bequeath the canvas to our new incarnation

            newSvgCanvas = putSvgIntoCanvas(svg, canvas)
            node = canvas.find_withtag(currentNode)[0]  if currentNode
            newSvgCanvas.select(node)  if node
        end
```

Notice our current SvgCanvas object will now only live until its methods return. This method passes stewardship of our TkCanvas into putSvgIntoCanvas(), which generates a successor SvgCanvas. Because our architecture, though imperfect, is decoupled, neither our production code nor our tests need any further changes to either insert this SvgCanvas into putSvgIntoCanvas(), or to inform everyone that SvgCanvas is now a new object. Block closures isolate SvgCanvas, so nobody cares who services the TkCanvas features.

Classes hold behavior, not data.

The code to edit nodes received a little upgrade, to handle nodes with linefeeds. You can't type them in yet, but you can copy and paste them in:

```
        def removeEditor(canvas)
            @plug.remove()
            @edit.after_idle{ @edit.destroy() }
```

```
            canvas.focus()  #   otherwise focus falls back to the frame
        end

        def saveNewText(canvas)
            tags = @lastItem.gettags()
            text = @edit.value.split(/\n/).join('\\n')  #  enable linefeeds
            tags[1] = [text]  #  [] so the tags record spaces
            removeEditor(canvas)
            @lastItem.configure('tags', tags)
            upgradeCanvas(canvas)
        end

        def editCurrentNode(canvas)
            return if not @lastItem
            @edit = TkText.new()
            x1,y1,x2,y2 = @lastItem.bbox()
            nodeText = @lastItem.gettags()[1]
            @edit.value = nodeText.split(/\\n/).join("\n")  if nodeText

            @plug = TkcWindow.new(
                        canvas, x1, y1,
                        :window => @edit,
                        :width  => x2-x1,
                        :height => y2-y1,
                        :anchor => 'nw'
                        )

         #  so user can start typing without "click-into" the edit field
            @edit.focus()

            @edit.bind('Key-Escape') {
                removeEditor(canvas)
                }
            @edit.bind('Key-Return') {
                saveNewText(canvas)
                }
        end
...
    end
```

Those features required shaking and jiggling to fall into place. For example, if you create a `TkEdit` field, then remove its `TkcWindow` node, it still exists in memory, and the canvas cannot receive the keyboard `focus()`. However, if the `TkEdit` calls a bound event handler which calls `@edit.destroy()`, control flow returns from the bound event into a dead object, and Tk crashes. The fix for that detail is to pass a block closure to Tk tell it to wait until all other events have dispatched, then destroy this window:

```
    @edit.after_idle{ @edit.destroy() }
```

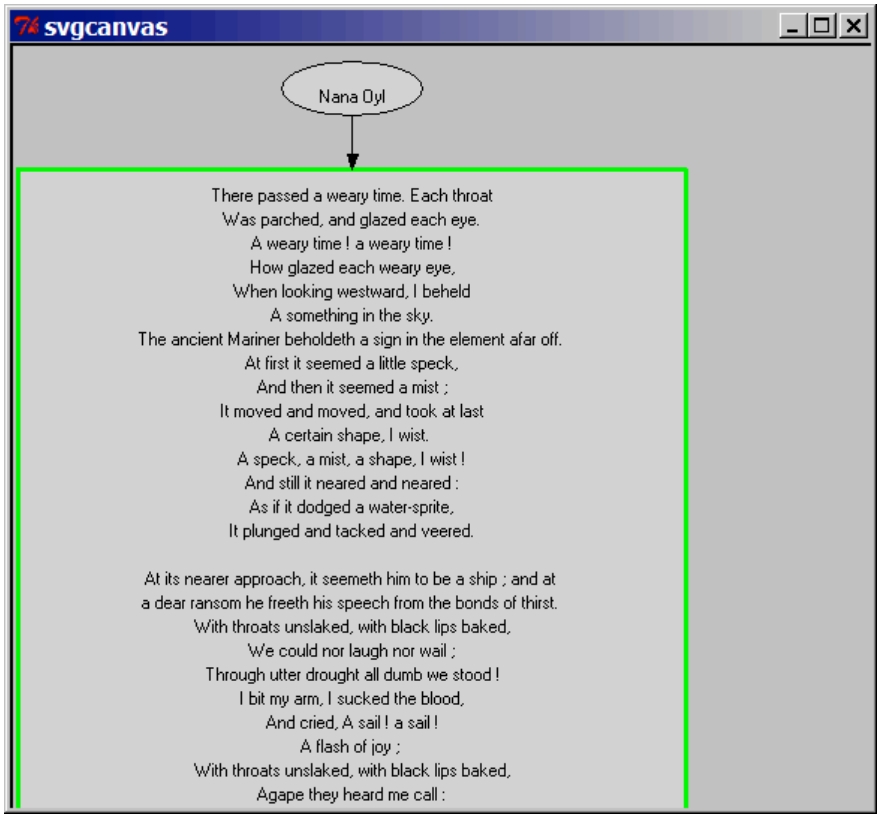
All those code details permit *Temporary Interactive Tests* that show our focus and edit behaviors working properly. This new test confirms our nodes can store and display text with linefeeds:

```
        def test_editWithLinefeeds()
            aCanvas = generateNanaOyl()
            oval, box, arrow, text1, text2, line = @canvas.find_all
            click(box)
            click(@canvas, 'Key-F2')
```

```
    edit = @canvas.find_all[6]
    edit.window.value = "I\nfeed\nlines"
    click(edit.window, 'Key-Return')

    oval, box, arrow, text1, text2, text3, text4, line =
                                          @canvas.find_all()
    assert_equal('I',     text2.text)
    assert_equal('feed',  text3.text)
    assert_equal('lines', text4.text)
    click(@canvas, 'Key-F2')  #   this fails unless the same node reselected
    edit = @canvas.find_all[8]
    assert_equal("I\nfeed\nlines", edit.window.value)
    click(edit.window, 'Key-Escape')  #   also test canceling an edit
    maybeMainloop()
end
```

The next feature will require more explicit tests on more interactive details. To close the current feature with a manual test, recall the purpose of regenerating the DOT notation was to allow `dot` to rebalance our layout when text changes size. So, turn on `maybeMainloop(true)`, click on Castor Oyl's rectangle, tap <F2>, and…

…paste in Part III of "The Rime of the Ancient Mariner", by Samuel Taylor Coleridge. Then hit <Enter>:



When you input twisted data into dynamic graphical user interfaces, you can create abstract art. Reviewing it reveals whether each programmed feature works as expected. That rectangle is large enough to contain all that poetry, its text did not turn into trash, Nana Oyl still links to it with an arrow, etc.

When you encounter (or deliberately produce) such a situation, save a picture of it, print it out, pin it to a collection on your bulletin board, and review it for usability. Our `TkEdit` field should, for example, limit text to 50 characters. Or our canvas's frame should have scroll bars.

Edit fields support linefeeds (pasted in) to enable biographical details in our Family Tree. Going forward, we might need to limit our text length. All our strings are variable-length, but our tree's esthetics must be predictable.

Now that users can edit node contents, we must allow them to interactively manipulate links.

Rubber Bands

This Case Study will demonstrate one more kind of feature. Our project should eventually add a context menu saying "Add Mother, Add Child", etc. And it should have a `main()`, and Load and Save features. You have already seen how to research Setting and Getting properties for various GUI controls, so menus shouldn't be too hard. Our last example is dynamic, interactive, and risky.

This is a rubber band:

After the (enhanced) mouse pointer clicks on a node, holds down the first button, and drags off, a line with an arrowhead links the clicked spot to the mouse. It stretches to wherever the mouse pointer goes. If the pointer goes over a new node that could become a child of Nana Oyl, such as Olive Oyl, we will give that node a temporary selection emphasis. Dropping the arrow—by releasing the mouse button—would declare that Nana Oyl begat Olive Oyl. The program will then update its data model, and re-render the canvas with a permanent & correctly formatted arrow.

(Note; in that detail, TkCanvas supplied the smallest arrowhead. `dot` supplied the larger one, pointing to Castor Oyl.)

To learn how the code observes a dragging mouse, write a *Temporary Interactive Test*:

```
def test_rubberBand()
    generateNanaOyl()
    oval = @canvas.find_all[0]

    oval.bind('B1-Motion') { |e|  p([e.x, e.y])  }

    maybeMainloop(true)
end
```

Run that, click the first mouse button on the Nana Oyl node, keep the button down, and move the mouse anywhere. The console will stream many x,y locations:

```
…
    [92, 31]
    [94, 32]
    [95, 33]
```

```
[97, 33]
[98, 34]
[99, 35]
[101, 35]
[102, 35]
[103, 36]
[104, 36]
[105, 36]
[107, 37]
```
…

As you drag, notice the oval has "captured" the mouse. Mouse movement events continue to trigger that callback, even after the mouse is no longer over that oval. Releasing the mouse button terminates the stream of events.

Tk's `.bind()` system collates several different kinds of primitive events—mouse clicks, movements, and releases—into a single high-level event that closely matches the usability envelope we envision for rubber bands. If your GUI Toolkit does not, you should assemble these primitives into a reusable system, decoupled from your rubber bands, before proceeding.

Releasing the mouse button produces the absence of an event. We need the presence of an event to trigger our release behavior, to drop that arrow:

```
def test_rubberBand()
    generateNanaOyl()
    oval = @canvas.find_all()[0]

    oval.bind('B1-Motion') { |e| p([e.x, e.y]) }

   oval.bind('ButtonRelease-1') {
        puts("released")
        }
    maybeMainloop(true)
end
```

Run that, click on Nana Oyl, drag the mouse off, and drop. The console emits a stream of mouse locations, and then "`released`".

The first mouse location is `[87, 26]`. This point lies within Nana Oyl's oval. We will use this to *Simulate User Input* and write a test that expects a rubber band.

Our `click()` fixture, as written, cannot yet take `x`,`y` coordinates. This little test shows how to pass them, but we won't use this technique:

```
def test_passMouseCoordinates()
    generateNanaOyl()
    oval = @canvas.find_all[0]
    cb_proc = proc{ |x,y| p([x,y]) }
    oval.bind('B1-Motion', cb_proc, '%x %y')
    cb_entry = oval.bindinfo('B1-Motion')[0][0]
    cb_entry.call('87', '26')
end
```

Run that, and predict `[87, 26]` will appear on the console.

A rubber band is an animation. If we *Regulate the Event Queue* correctly, we can create an animated visual inspection. The sub-chapter beginning on page 271 shows how ImageMagick can capture such a display as an animated GIF file, for storage and remote review. We won't go that far, just for a rubber band.

Tk permits us to use `.event_generate()`, but only if our window displays, at least briefly, on the screen. This project used silent test cases until we found a reason to flicker our windows. This test shows how things work:

```
def test_mockEvent()
    generateNanaOyl()
    oval = @canvas.find_all[0]
    x = 0
    y = 0

    oval.bind('B1-Motion') { |event|
        x = event.x
        y = event.y
        }
    Tk.update()  #   run the event queue until empty
    oval.event_generate('Enter',     :x => 87, :y => 26)
    oval.event_generate('B1-Motion', :x => 87, :y => 26)
    assert_equal(87, x)
    assert_equal(26, y)
    maybeMainloop()
end
```

Tk provides `update()` to dispatch all the pending events in its queue. We call that to display our target window on the screen, and cook the internal variables that `.event_generate()` depends on.

However, TkCanvas cannot exercise an item's `'B1-Motion'` event unless it sees `'Enter'` first. That's not the keystroke <Enter>, it's the mouse flying over the boundary into an item. *Simulating User Input* always causes friction with how GUI Toolkits interpret real input. If you get stuck with your Toolkit, try another way. For our current project, `Tk.update()` followed by our `click()` fixture would have worked fine.

Now fixturize the fruit of our research:

```
def mockEvent(what, how, x, y)
    Tk.update()  #   run the event queue until empty
    what.event_generate(how,     :x => x, :y => y)
end

def test_mockEvent()
    generateNanaOyl()
    oval = @canvas.find_all[0]
    x = 0
    y = 0

    oval.bind('B1-Motion') { |event|
        x = event.x
        y = event.y
        }
    mockEvent(oval, 'Enter',     87, 26)
    mockEvent(oval, 'B1-Motion', 87, 26)
    assert_equal(87, x)
    assert_equal(26, y)
    maybeMainloop()
end
```

When that passes, it's time to animate rubber bands.

Motion Capture

First, write this test to capture a sample of your input:

```
    def test_recordMotion()
        generateNanaOyl()
        oval = @canvas.find_all[0]

        oval.bind('B1-Motion') { |event|
            puts([event.x, event.y].inspect() + ',')
            }
        maybeMainloop(true)
    end
```

Run that, click on Nana Oyl's oval, drag the mouse out, and drop. The console fills up with `[86, 34], [88, 35], [91, 36], [94, 37],` etc. Those coordinates are correctly formatted for a Ruby array. Copy them out and paste them into the next test:

```
    def test_mockUser()
        aCanvas = generateNanaOyl()
        oval = @canvas.find_all[0]

        motion = [ [ 86, 34], [ 88, 35], [ 91, 36], [ 94, 37],
                   [ 98, 39], [102, 41], [107, 43], [112, 44],
                   [116, 46], [120, 48], [122, 49], [125, 50],
                   [126, 50], [127, 50], [127, 51], ]

        motion.each() do |x, y|
            mockEvent(oval, 'B1-Motion', x, y)
            end

        maybeMainloop()
    end
```

The statement `motion.each() do |x, y|` … `end` is Ruby's way of saying this, in C++:

```
for (int index(0); index < motion.size(); ++index)
  {
  int x = motion[index].x;
  int y = motion[index].y;
…
  }
```

Ruby makes the common task of iteration more elegant by partitioning it into slightly different abstractions.

However, when you run the tests we just wrote, it appears to do nothing. We have added no feature yet. Incrementally add to the test assertions that expect a rubber band to perform correctly, and pass each test. The code turns out like this:

```
    class EllipseBuilder < Builder
…
        def newThing(anSvgCanvas, canvas, svg)
…
            element.bind('Button-1') {
                anSvgCanvas.select(element)
                }
            element.bind('B1-Motion') { |event|
```

```
                anSvgCanvas.drawRubberBand(canvas, event.x, event.y)
                }
            element.bind('ButtonRelease-1') { |event|
                anSvgCanvas.dropRubberBand(canvas, event.x, event.y)
                }
        end
    end
```

As noted elsewhere, `EllipseBuilder` and `PolygonBuilder` duplicate some behaviors. All `<ellipse>` SVG tags are nodes, but only some `<polygon>` tags are nodes. And all nodes are interactive, so both these classes share the "family member" concept. These classes share a latent "`GraphNode`" class, waiting to emerge. If these technical issues make more problems than adding the same `.drawRubberBand()` call two places, we should take care of them.

```
    class PolygonBuilder < Builder
…
        def newThing(anSvgCanvas, canvas, svg)
…
            if 'edge' != nodeClass then  #  don't register arrowheads
                nodeName = anSvgCanvas.registerNode(element, svg)
                anSvgCanvas.addNodeName(nodeName)

                element.bind('Button-1') {
                    anSvgCanvas.select(element)
                    }
                element.bind('B1-Motion') {  |event|
                    anSvgCanvas.drawRubberBand(canvas, event.x, event.y)
                    }
                element.bind('ButtonRelease-1') { |event|
                    anSvgCanvas.dropRubberBand(canvas, event.x, event.y)
                    }
            end
…
        end
    end
```

The canvas now tracks a rubber band, and erases it when the user releases the mouse button:

```
    class SvgCanvas
…
        def getRubberBand()
            return @rubberBand
        end

        def drawRubberBand(canvas, x, y)
            if not @rubberBand then
                @rubberBand = TkcLine.new(canvas, [x,y,x,y])
                @rubberBand.arrow = 'last'
            end

            x1,y1,scratch,scratch = @rubberBand.coords()
            @rubberBand.coords(x1, y1, x, y)
        end

        def dropRubberBand(canvas, x, y)
            @rubberBand.remove  if @rubberBand
            @rubberBand = nil
```

```
        end
...
    end
```

And the test forced them all to work:

```
    def test_mockUser()
        aCanvas = generateNanaOyl()
        oval = @canvas.find_all[0]

        motion = [ [ 86, 34], [ 88, 35], [ 91, 36], [ 94, 37],
                   [ 98, 39], [102, 41], [107, 43], [112, 44],
                   [116, 46], [120, 48], [122, 49], [125, 50],
                   [126, 50], [127, 50], [127, 51], ]

        assert_nil(aCanvas.getRubberBand())
        mockEvent(oval, 'Enter', 86, 34)
        mockEvent(oval, 'Button-1', 86, 34)

        motion.each do |x, y|
            mockEvent(oval, 'B1-Motion', x, y)
            rb = aCanvas.getRubberBand()
            assert_kind_of(TkcLine, rb)
            expect = [86, 34, x, y]
            assert_equal(expect, rb.coords)
            assert_equal('last', rb.arrow)
            end

        mockEvent(oval, 'ButtonRelease-1', 127, 51)
        assert_nil(aCanvas.getRubberBand())
        Tk.update()
        maybeMainloop()
    end
```

Those `Tk.update()` calls produce a funny side-effect. Each time the tests run, this one displays its window and plays an animation of its own simulation. If we felt this feature needed *Broadband Feedback*, we could collect screen captures of those animations, for later review.

("Yo", by the way, is American slang for "Hello World".)

Manually testing the canvas, with `maybeMainloop(true)`, reveals correct rubber band behavior, but nothing happens when we drop the arrowhead onto another node.

Secondary Selection Emphasis

Eventually, the canvas will support a New Node menu command, to create an orphan individual with no family ties. Then, dragging a rubber band from a parent to the new node will adopt it. But to test rubber bands, and secondary selection emphases, we skip the New Node command. Naturally, between iterations, we proceed in business value order, but engineers within an iteration may select technical risk order. Your Tk tutorials probably cover menus in enough detail for you to learn how to Set and Get their properties. Our highest risk feature is still our rubber bands. So we need to test a canvas that already contains an orphan node, as if a user had created it with the potential New Node command.

We simply hotwire our test to assemble its data as if the user had invoked a feature that does not exist yet:

```
def generateNanaOyl(hotWire = '')
    svg = generateSvg('digraph aGraph {
        node [style=filled];
        person1 [label="Nana Oyl"];
        person2 [label="Castor Oyl", shape=box];
        ' + hotWire + '
        person1 -> person2; }')


    return putSvgIntoCanvas(svg, @canvas)
end

def test_recordMotion()
    generateNanaOyl("person3 [label=\"Olive Oyl\"];\n")
    oval = @canvas.find_all[0]

    oval.bind_append('B1-Motion') { |event|
        puts([event.x, event.y].inspect() + ',')
        }
    maybeMainloop(true)
end
```

Now run that *Temporary Interactive Test*, see this…

…and drag from Nana Oyl to Olive Oyl. You'll get a long motion capture, `[86, 25], [87, 25], [88, 25],` etc. I'l cut the middle nodes out so our next test isn't so long.

Now the test must simulate a user dragging a rubber band from Nana Oyl to Olive Oyl. First write it, and watch it dragging the arrow to its target:

```
def test_mockUserDragAndDrop()
    aCanvas = generateNanaOyl("person3 [label=\"Olive Oyl\"];\n")
    oval = @canvas.find_all[0]
    count   = @canvas.find_all.size()

    motion = [ [ 86, 25], [ 91, 27], [ 92, 27], [ 98, 28],
               [100, 28], [104, 29], [105, 29], [111, 29],
               [112, 29], [118, 30], [119, 30], [121, 30],
               [126, 30], [127, 30], [128, 29], [131, 29],
               [137, 27], [139, 26], [140, 26], ]

    mockEvent(oval, 'Enter',    86, 25)
    mockEvent(oval, 'Button-1', 86, 25)

    motion.each do |x, y|
        mockEvent(oval, 'B1-Motion', x, y)
        end

    mockEvent(oval, 'ButtonRelease-1', 140, 26)
    Tk.update()
    maybeMainloop()
end
```

When the rubber band's arrowhead goes over Olive Oyl, that node will display a secondary selection emphasis. So add lines that find Olive Oyl's node, and assert that it is thick, green, and dashed:

```
        motion.each do |x, y|
            mockEvent(oval, 'B1-Motion', x, y)
            end

        olive = @canvas.find_all[1]
        assert_equal(3.0,     olive.width  )
        assert_equal('green', olive.outline)
        assert_equal([7],     olive.dash   )
```

That will look like this, when we finish the code:

I also authored a change to the arrow, to give it a thick green dashed line, too, to match the target:

```
    def drawRubberBand(canvas, x, y)
        if not @rubberBand then
            @rubberBand = TkcLine.new(canvas, [x,y,x,y])
            @rubberBand.arrow = 'last'
            @rubberBand.configure('fill', 'green')
            @rubberBand.configure('width', 3.0)
            @rubberBand.configure('dash', 0x0f)
        end
…
    end
```

Okay, we are ready to declare that Nana is Olive's mom.

Begat

Add lines to the test case that assert when the arrow drops on that node, we rebuild the canvas, with a new line from Nana Oyl to Olive Oyl:

```
        mockEvent(oval, 'ButtonRelease-1', 140, 26)
        Tk.update()
        assert_equal(count + 2, @canvas.find_all.size())
        assert_equal('person1->person3', @canvas.find_all[8].gettags[2])
```

This is a serious new feature. It requires the ability to detect which node is under an arrowhead, the ability to roll-back and do nothing if the user moves off that node, and the ability to regenerate DOT notation with a new command inside it. All these code features should have come from Child Tests, and should get more refactoring, but the end of this chapter is in sight.

To find a node under the arrowhead:

```
    def findNodeUnderArrowhead(canvas, x, y)
        found = canvas.find_overlapping(x, y, x + 1, y + 1)

        found.each() do |item|
            tags = item.gettags()

            if tags.size() >= 2 and
               tags[0] =~ /^node/ and
               tags[2] =~ /^person/ then
                return item
```

```
            end
        end

        return nil
    end
```

After stretching a rubber band, look for a node under it, and set its secondary selection emphasis:

```
    def drawRubberBand(canvas, x, y)
        if not @rubberBand then
…
        end

        x1,y1,scratch,scratch = @rubberBand.coords()
        @rubberBand.coords(x1, y1, x, y)

        found = findNodeUnderArrowhead(canvas, x, y)
        setPartialSelection(found)
    end
```

This either turns on the emphasis or turns it off, depending if the arrowhead is coming or going:

```
    def setPartialSelection(item)

        return  if not getRubberBand()

        return  if @lastItem and item and
                   item.gettags() == @lastItem.gettags()

     #  TODO if I am elegible...

        if item then
            @lastPartialSelection = item
            item.configure('width'  , 3.0     )
            item.configure('outline', 'green' )
            item.configure('dash'   , 0x07    )
            @lastPartialSelection = item
        elsif @lastPartialSelection then
            item = @lastPartialSelection
            item.configure('width'  , 1.0     )
            item.configure('outline', 'black' )
            item.configure('dash'   , ''      )
        end
    end
```

The TODO appeared because we need a Logic Layer. It should provide a method to detect nodes not eligible to become someone's child. setPartialSelection() should use that method to refuse the secondary selection emphasis when dragging over an unqualified node.

When a rubber band drops, check if a node has the secondary selection emphasis (a "partial selection"), and rebuild the canvas:

```
    def dropRubberBand(canvas, x, y)
        @rubberBand.remove  if @rubberBand
        @rubberBand = nil

        if not maybeLinkToPartialSelection(canvas) then
```

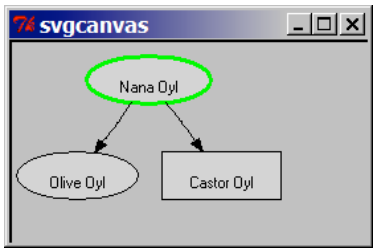
```
            setPartialSelection(nil)
        end  #  else items like oval no longer exist
    end
```

Rebuilding the canvas requires this slightly crufty function:

```
    def maybeLinkToPartialSelection(canvas)
        return false  if not @lastPartialSelection
      #  maybe not

        fromName = @lastItem.gettags()[2]
          toName = @lastPartialSelection.gettags()[2]

      #  convert canvas to DOT notation

        dot = "digraph aGraph {\n" +
            "node [style=filled];\n"

        canvas.find_all().each() do |item|
            dot += writeDOTstatement(item)
            end

      #  add the new link to it

        dot += fromName + '->' + toName
        dot += "}"

      #  and rebuild the canvas

        svg = ''
        open('scratch.dot', 'w') do |f| f.write(dot) end
        system 'dot -Tsvg scratch.dot -o graph.svg'
        open('graph.svg', 'r') do |f| svg = f.read()  end

      #  preserve current node selection, then croak all items

        currentNode = @lastItem.gettags[2]  if @lastItem
        canvas.remove('all')

      #  bequeath the canvas to our new incarnation

        newSvgCanvas = putSvgIntoCanvas(svg, canvas)
        node = canvas.find_withtag(currentNode)[0]  if currentNode
        newSvgCanvas.select(node)  if node
        return true
    end
```

That looks bad, but you should find cleaning it up very easy. It should have reused methods like `.upgradeCanvas()` and `.canvasToSvg()`. They looked short and clean, but they were not reusable. Each contained subtle assumptions that looked harmless when they were written. Reuse is hard to plan, and easy to refactor. I copied all those methods' guts out and pasted them into this new method. Only a few Extract Method Refactors are needed to produce truly reusable methods.

This Case Study is finished:



All the remaining refactors and new features will be easy, using the fixtures that these few hard features spawned.

To Do

The previous Case Study wrote display-only code, and illustrated `maybeMainloop(true)` *Regulating the Event Queue*. And it illustrated the regular TFP cycle, including refactoring to clean up design and make subsequent features easier.

This chapter added features in order of technical risk, not business value. (That's okay within an iteration. Sort features by business priority between iterations.) A real project would add more tests, refactor the code to clean up its run-on methods, and fix a number of esthetic issues. For example, if a Family Tree grew too large, the canvas should grow scroll-bars. These should scroll a viewport over a large virtual canvas. Using <Tab> or an arrow key to move the selection emphasis should scroll to keep that emphasis on-screen. When code rebuilds a canvas, and restores the selection emphasis to a node, it should center in the viewport.

A number of potential features appear in Chapter 19: *Exercises*, on page 460.

Conclusion

This chapter revealed the temptations of manually operating a GUI while adding interactive features. However, last chapter's fixture, `maybeMainloop(true)`, enabled *Temporary Interactive Tests* that saved us. We wrote interactive features by writing test and production code interactively.

Chapter 7: *NanoCppUnit*

Many teams think of the venerable language C++, and the Windows Software Development Kit (SDK), as cruft-seeking missiles. Hard to steer, fast, and likely to rapidly create a big mess.

This first of three Case Studies will launch a small dialog, and build its test rig from scratch, using techniques that compete easily with other platforms. The application is the most common GUI: A data entry form with a few fields to display and edit data.

Our new test rig, and Windows Template Library (WTL), wrap many low-level details, making development safe and rapid. The next Case Study, *Model View Controller* (on page 194), will refactor the simple data entry features into the seed of a healthy and flexible layered design. The final Case Study, *Broadband Feedback* (on page 217), will upgrade esthetics until one of our project's skins looks like that picture.

To illustrate the shortest path to full-featured *Fault Navigation*, we use powerful C++ techniques that work with any MS Windows IDE. To keep the code clean, while intercepting any error, we use the Visual C++ `<comdef.h>` library, and related systems. They constrain Component Object Model (COM), convert errors into C++ exceptions, and manage COM data objects' lifetimes, preventing leaks.

This chapter's source grows with the phaseless rhythm of Test-First Programming. That mixes some low-level details up with high-level design notions, especially early when everything bootstraps. Here's the sequence of events:

- Page 167: Begin a dialog with a "First Name" field, and a prototypical test case.
- 170: Upgrade the test to provide *Fault Navigation*.
- 172: Introduce COM and MSXML.
- 174: Handle their error events aggressively.
- 181: Add "Last Name" and "Address 1" fields.
- 183: Roll all the test cases into a test runner.
- 186: Discuss differences between common C++ fallacies and mine.

Modern GUIs frequently use "Object Request Brokers" (ORBs) to bond controls to each other and to other modules. ActiveX Template Library hides their low-level details in clean wrappers. This Case Study follows the advice of *Accelerated C++: Practical Programming by*

Example, by Koenig & Moo. We manage text strings with `std::string` objects, and manage lists of objects with `std::list` containers. We don't copy strings into raw char arrays, or hand-code linked lists to store objects. And we don't clutter our code with excessive calls to COM mechanics.

Those of you burned by excessively crufty "C-style C++" code, or old-fashioned, fragile, redundant "OLE" code, will find this source refreshing.

Programmers need to learn to incorporate *Fault Navigation* into any editor. This Case Study builds a test runner directly into any IDE. Our test rig passes diagnostic strings into `OutputDebugString()`. That displays them in an Output Panel. This technique enables many IDEs to function as test runners, without extending the IDE's features.

If you want to use CppUnit, or another language and its test rig, to play along with this Case Study, skim this area to see what our early dialog and XML input look like, bootstrap your own project, and engage test-first after page 181.

Along the way, a few rules, which C++ engineers learned are sacred, get bent.

Visual Studio and Friends

Visual C++ is Microsoft's C++ editor, compiler, and debugger. Its projects, by default, compile in two modes, Debug and Release. The former activates a number of debugging features, compiled by the `_DEBUG` macro condition, and the latter compiles an optimized production build.

Debug mode's features defend Release mode from bugs, so our tests run here. Eventually, our program will take a command line argument, `--test`, conditionally compiled only in Debug mode. That would switch between a test run or a production run.

Those techniques provide One Button Testing. The VC++ Go command, <F5>, compiles and runs the current program, which will optionally test itself. This strategy avoids extra effort, such as manually invoking a test rig.

Bjarne Stroustrup invented C++ to provide the popular but low-level C language with the keywords `virtual`, `template` and `operator`. Those enable Object Oriented techniques with minimal runtime overhead. C is "portable assembler", and C++ is "portable OO assembler".

Microsoft developed Component Object Model to permit any language to manipulate objects inside a DLL as high-level objects, not as low-level C style functions. Any similarity to CORBA, another popular ORB, must be just a shocking coincidence.

(I searched the Internet for an early paper about COM, Anthony Williams's "Object Architecture: Dealing with the Unknown," and only discovered a post claiming it was originally written in Klingon. We are undoubtedly in Andy's debt for translating it into Microsoft-speak.)

A client module uses COM to create and manipulate high-level objects inside another module. Such objects exist within their own language system—as defined by the Interface Definition Language—so any programming language can adapt to them.

Microsoft Foundation Classes wrap MS Windows objects' handles with classes, and permit a Class Wizard to manipulate the links between a program's resource segment and its GUI class methods. The Wizard similarly wraps COM objects and their host sites. MFC tuned for desktop applications, where programs are aqlready very large.

When the Internet became popular, requiring very lightweight GUI controls, MS re-wrote their C++ wrapper for COM, preferring templates to virtual methods. The ActiveX Template Library, to support the windows that COM objects might raise, also wrapped some of MS Windows' core window objects.

Nenad Stefanovic, a Microsoft engineer, extended the library into much of MFC's territory, producing Windows Template Library. The best documentation for this skunkworks appears here: `http://www.codeproject.com/wtl`

C++ in a Nutshell

This project is going to survey some thorny low-level issues, then rapidly lead to a high-level design. Solutions for these issues follow principles that educate users of all languages. To keep everyone on the same page, C++ itself needs a quick introduction. (To "play along" with this Case Study, first learn C++ and WTL from textbooks. This Nutshell only provides enough concepts to read a little C++—not write it!)

C++ is a statically typed language. It has no latent "`Object`" base class to supply default behaviors to all objects. C++ supports Object Oriented techniques, but not everything in C++ is an object.

All C++ entities have a declaration and a definition. C++ uses a compiling and linking system that compiles many declarations and a few definitions together into one "object code file". Then many of these link together to form a complete program.

Only source statements compiled below a declaration can use its identifiers. Developers can declare and define classes and functions simultaneously (the preferred way), or they can move declarations to separate files. This Case Study will show tests followed by the code that passes them, but that code, or its declarations, must in source appear above their tests.

C++ supports very large Object Oriented applications when modules typesafely compile remote modules' interface declarations, without the burden of recompiling their implementation definitions.

Language Law: C++ supports both methods and modules, but has no single explicit keyword for either. Programmers assemble methods and modules from primitive facilities. Roughly speaking, a method is a `virtual` class member function, and a module is a translation unit, typically based on compiling one `.cpp` file.

The command to import a module's declarations is `#include`, and the command to import an identifier from that module is `using`. The Hello World project for C++ is:

```
#include <iostream>   //  copy in declarations, and some definitions

    using std::cout;  //  meaning Console OUTput
    using std::endl;  //  meaning END of Line

int main()
{
    cout << "Hello World!" << endl;
    return 0;
}
```

The function `main()` returns a `0`, which tells the environment that this program ran without errors. Eventually, failing tests will return `1`.

The number `0` has a weak type, usually `int`eger. (It may also become a `NULL` pointer, or `false`, depending on context.) All C++ functions declare their return type, or they declare `void` to return nothing.

In my Uncommon Style Guide, functions' return types are less important than the functions' names. Indenting the return values, like this…

```
    int
```

```
    main()
    {
…
    }
```

…cleans the left column, so scanning it reveals only function names. This style becomes very easy to read—especially as function types get long.

Applying the Object Method Refactor to the Hello World application yields this class:

```
    using std::ostream;

        class
    WorldGreeter
    {
      public:

        void
    greet(ostream &out)
        {
        out << "Hello World!" << endl;
        }
    };

        int
    main()
    {
        WorldGreeter aGreeter;
        aGreeter.greet(cout);
        return 0;
    }
```

The console stream object, `cout`, passes into `greet()` by reference. `greet()` takes a reference (`&`) to an `ostream`, which is one of `cout`'s classes' base classes. `ostream` and many classes derived from it use `<<` to insert data into their streams.

This stream insertion concept makes the following chapters easier to understand. C++ permits operator overloading, meaning programmers specify what some operators do. The operator `<<`, applied to integers, shifts their bits, so `2 << 3 == 16`.

Shifting objects makes no sense, so `<<` is free to overload for stream objects. Refactoring our "Hello World" project, one more time, illustrates this principle:

```
        class
    WorldGreeter
    {
      public:

        void
    greet(ostream &out) const
        {
        out << "Hello World!";
        }
    };

        ostream &
    operator<<(ostream &out, WorldGreeter const &aGreeter)
    {
        aGreeter.greet(out);
        return out;
```

```
}

    int
main()
{
    WorldGreeter aGreeter;
    cout << aGreeter << endl;
    return 0;
}
```

The ability to insert any object, as text, into a stream, is more than syntactic sugar. When `templates` and macros call `operator<<`, they can use any argument type that has such an operator. The Standard Library defines `operator<<` for all the primitive data types—`int`, `double`, `std::string`, etc.—and programmers define one for each object they need. (Page 175 includes code that defines `operator<<()` for `_bstr_t` string types.) Note the above `operator<<` is part of `WorldGreeter`'s interface, but is not a member of the `WorldGreeter` class. C++ types work in mysterious ways.

In C++, classes are not objects. This makes passing a class into a method, as a parameter, problematic. (By contrast, page 97□ displays Ruby code passing the classes `TkcOval` and `TkcText` into the method `constructor()` without a second thought.) C++ compiles all class details into various opcodes, wherever they occur, so a running C++ program has no single entity to refer to as a class.

C++ programs use `templates` to merge types. Each template instantiation expands a class-like definition into a set of classes, each varying by some type or integer. For example, the C++ Standard Library provides a `template` called `std::basic_string<>`. Supplying this with a character type provides string types of any needed width:

```
namespace std {
    typedef basic_string<char> string;      //   8-bits
    typedef basic_string<wchar_t> wstring;  //  16-bits
    }
```

The keyword `namespace` is one of the language primitives that collude to create modules. Identifiers from the C++ Standard Library often begin with `std::`, to distinguish their source module. A `namespace` is a prefix (`std`) that must precede all uses of identifiers declared within their scope (`string`, `wstring`). The scope operator, `::`, links them.

The keyword `typedef` stores a type declaration inside a single name. Here, it instantiates `templates` into their target classes. The sample creates the class `std::string`, to contain 8-bit character strings, and the class `std::wstring`, to contain 16-bit character strings.

To avoid writing `std::` all over the place, insert a `namespace`'s identifiers into your scope with a few `using` declarations:

```
#include <string>
#include <sstream>
#include <iostream>

using std::string;
using std::stringstream;
using std::cout;  //  console output stream
using std::endl;  //  line feed object
```

C++ macros are light secondary language that provide some important features. Here's a macro, using some of those stream things:

```
#define db(x_) cout << (x_) << endl
```

Macros substitute text before the type-sensitive phase of compilation. Put another way, if you can't think how to do something directly, you can often write a macro to do it. (Those of you who know C++ well enough to abhor that advice are requested to attempt the following techniques using any other C++ facilities.)

Below that macro, if you write `db(5)`, you get `cout << (5) << endl`. A rote text substitution, without awareness of `5`'s meaning. It's just raw text. The `cout << (5)` then inserts `5`, typesafely, into the console stream, so you can see it on a command prompt. The `<< endl` part delivers a linefeed, and flushes that console stream's buffer.

So far, a small templated function could have performed the same feat. Use macros for the stringerizer operator, `#`:

```
#define db(x_) cout << #x_ ": " << (x_) << endl
```

"`db`" stands for debug. If `q == 5`, and if `q` is suspicious, we can trace `q`'s name and value to the console using `db(q)`. That outputs "`q: 5\n`". This macro makes all trace statements easy to write. The equivalent `printf()` statement, when `q` is an integer, would be `printf("q: %i\n", q)`. Our statement works for any type that supports an `operator<<`, and it outputs the name of `q` automatically, using `#x_`.

Suppose a long program run emitted a strange message, complaining about some obscure variable named `q`. Rapidly finding the line of code containing that `db(q)` call relieves stress:

```
#define db(x_)  do { \
    cout << __FILE__ << "(" << __LINE__ << ") : " \
    #x_ " = " << x_ << endl; } while (false)
```

That emits, roughly, "*C:\path\source.cpp*`(99) : q = 5\n`". All from one easily written `db(q)` call.

The `do` and `while(false)` keywords force the statement containing `db(q)` to finish with a healthy `;` semicolon. Macros need techniques, like extra parentheses, to avoid clashing with raw C++ statements.

On the ends of broken lines, the reverse solidi, `\`, join everything into one long macro. The `__FILE__` and `__LINE__` symbols are magic constants, defined by the C languages, which expand into the current name of the source file and line number.

Before launching this Case Study's main project, we add one more trick. The *Fault Navigation* Principle links bug-prevention systems to your IDE. When a program encounters syntax errors, faults, or trace statements, the same keystroke must navigate to any of them. This rule permits using an IDE as a Test Runner, reducing your keystroke burden.

This project soon enables *Fault Navigation* for assertions. Trace statements should also be navigable. Adding *Fault Navigation* to trace statements is good practice for the assertions.

Visual Studio surfs to errors using <F8>: Go To Output Window Next Location. The Windows SDK function to write text into the output panel, for this feature to read it and surf to an error, is `OutputDebugString()`.

Putting them all together yields this killer trace macro:

```
#define db(x_)  do { std::stringstream z;     \
        z << __FILE__ << "(" << __LINE__ << ") : " \
                #x_ " = " << x_ << endl;      \
        cout << z.str() << std::flush;        \
        OutputDebugStringA(z.str().c_str());  \
        } while (false)
```

That takes any argument, including expressions, which support `operator<<`. We will return to these techniques while exploring more *Fault Navigation* issues in C++.

`db(q)` pushes "*`C:\path\source.cpp(99) : q = 5\n`*" into the Output Debug panel. <F8> parses the file name and line number and navigates your editor directly to the line containing the `db(q)`.

These are major wins. Tracing with `db()` is very low cost for very high feedback.

C++ has flaws. But those of you inclined to dismiss it entirely are invited to write `db()`, with all these features, in so few lines, using your favorite language.

Starting an Application without a Wizard

Wizards start most MS Windows SDK projects. You can select an application window layout, with a few optional details. Use these to learn how these windows' inventors assemble their internal details.

This project is going to grow the most primitive test rig and application possible. Starting simpler than any Wizard allows us to control those details carefully.

This project uses Visual Studio 7 because we need `CString` to interact correctly with `.GetDlgItemText()` methods. This project could work with VS6 and fixed-length strings.

Open DevEnv.exe (VS7's editor), and select New → New Project → Win32 → Win32 Console Project. Don't use the wizard. The default linker settings will be `/SUBSYSTEM:CONSOLE`, not `/SUBSYSTEM:WINDOWS`.

Now write a test:

```
#include <assert.h>

    void
test_Dialog()
{
    ProjectDlg aDlg("<user><name>Ignatz</name></user>");
    aDlg.Create(HWND_TOP);
    CString name;
    aDlg.GetDlgItemText(IDC_EDIT_FIRST_NAME, name);
    assert("Ignatz" == name);
    aDlg.DestroyWindow();
}

    int
main()
{
    test_Dialog();
    return 0;
}
```

That assumes a lot, revealing our intent. We intend a dialog that reads a tiny snip of XML and presents an edit field containing "Ignatz". I marked undeclared identifiers in **`bold`**.

This little project will read and write an XML file, with a hard-coded schema. A real project would have a reason to read and write such files, and that reason would lead to a Logic Layer. This project will soon split out a Representation Layer that doesn't use the GUI directly. That, in turn, would insulate the Logic Layer, if it existed, from changes in the GUI's format and arrangement. We could, for example, move some data entry fields into another dialog, then change the "wiring" in the Representation Layer. Any Logic Layer need never know.

To compile our first test, we need identifiers like "`IDC_EDIT_FIRST_NAME`". MS Windows grants each program a data segment containing resources, including specifications for windows and controls. Developers author these segments by writing resource files. If we give our project the imaginative name "Project", then its resource file could be named "`Project.rc`", and the declarations for its interface would be "`Project.h`".

Programs extract elements from resource segments via integers. After we use the menu Project → Add Resource… to paint a little dialog resource, "`Project.h`" contains this…

```
#define IDD_PROJECT                     101
#define IDC_EDIT_FIRST_NAME            1001
```

…and "`Project.rc`" contains this:

```
IDD_PROJECT DIALOGEX 0, 0, 187, 94
STYLE DS_SETFONT | DS_MODALFRAME | DS_FIXEDSYS |
    WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Project"
FONT 8, "MS Shell Dlg", 400, 0, 0x1
BEGIN
    DEFPUSHBUTTON   "Okay",IDOK,130,7,50,14
    PUSHBUTTON      "Cancel",IDCANCEL,130,24,50,14
    LTEXT           "Name",IDC_STATIC,7,18,45,10
    EDITTEXT        IDC_EDIT_FIRST_NAME,
                        61,17,53,13,ES_AUTOHSCROLL
END
```

I spared all expense to make that dialog appear esthetic and beautiful. The Visual Studio Resource Editor displays it like this:

We will soon have a better way to review and improve it than the Resource Editor.

CDialogImpl<>

The minimum new code to compile that test:

```
#include <atlStr.h>
#include <atlApp.h>

    CAppModule _Module;

#include <atlWin.h>
#include <atlDlgs.h>
#include <assert.h>
#include "resource.h"

    class
ProjectDlg:
```

```
    public CDialogImpl<ProjectDlg>
{
  public:
    enum { IDD = IDD_PROJECT };

    BEGIN_MSG_MAP(ProjectDlg)
    END_MSG_MAP()

    ProjectDlg(char *xml) {}
};
```

`CDialogImpl<>` wraps dialogs. MS Windows operates windows and dialogs as internal objects, and gives us `HANDLE`s. That narrow interface decouples MS Window's private details from all their customers' programs. WTL wraps `HANDLE`s with classes that parallel the secret internal objects' identity and lifespan.

Now add lines to the dialog class to pass the test:

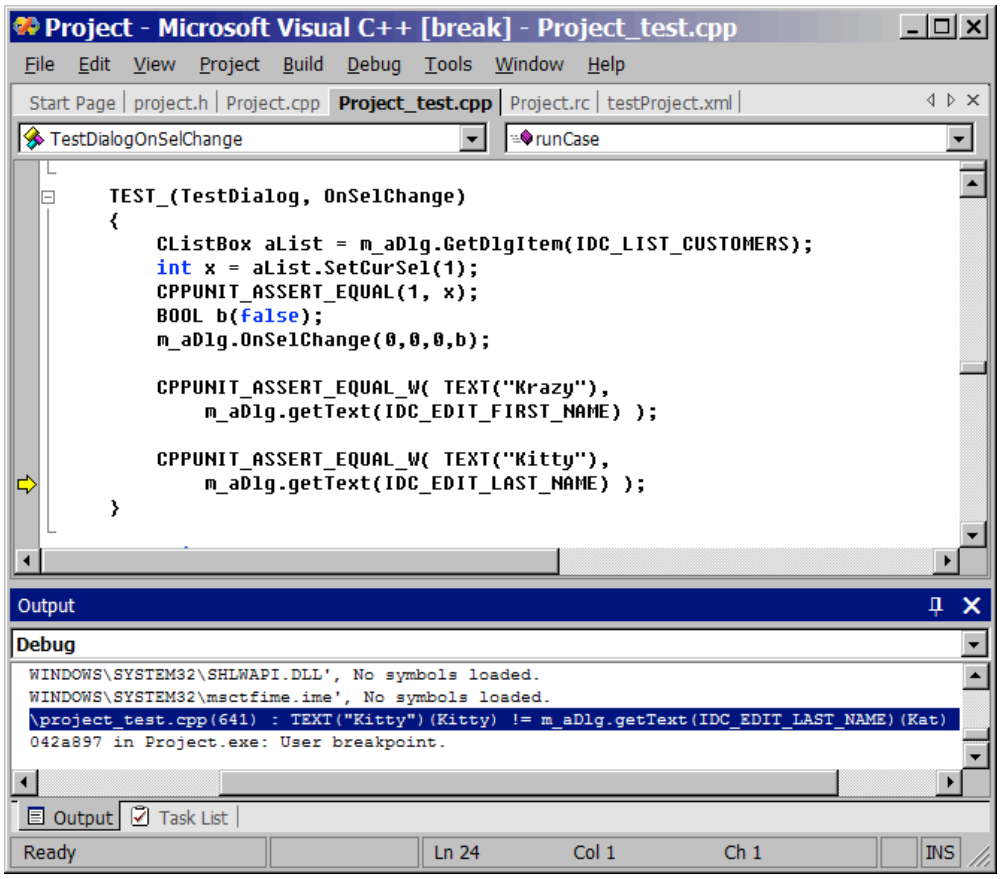
```
    class
ProjectDlg:
    public CDialogImpl<ProjectDlg>
{
  public:
    enum { IDD = IDD_PROJECT };

    BEGIN_MSG_MAP(ProjectDlg)
        MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
    END_MSG_MAP()

        LRESULT
    OnInitDialog(UINT, WPARAM, LPARAM, BOOL &)
        {
        SetDlgItemText(IDC_EDIT_FIRST_NAME, "Ignatz");
        return 0;
        }
    ProjectDlg(char *xml) {}
};
```

Notice we did *not* start our dialog by adding a bunch of private string members—`m_firstName`, `m_lastName`, `m_address1`, etc. We started with the external features that other application objects, and the user, need from dialog objects: Programming by Intention.

We are not surprised that `OnInitDialog()` lies. Previous chapters show tests and duplication removal forcing out such lies. Here, we'll first demonstrate better *Fault Navigation* than the C library `assert()` macro.

Visual C++ Fault Navigation

This is our goal:



When a test faults, the editor displays the assertion's complete verbiage:

```
...\project\project_test.cpp(641) :
   TEXT("Kitty")(Kitty) != m_aDlg.getText(IDC_EDIT_LAST_NAME)(Kat)
```

The error message tells us a lot about the situation; both the text of the input expressions, and their values. These features permit us to rapidly assess if a test failed for the correct reason.

If we decide to navigate to the file `project_test.cpp`, line 641, we need only tap the <F8> key. I did that before taking the screen shot, so the little yellow arrow in the left gutter points at the failing assertion. (It failed for the correct reason—because I wrote "Kitty" on it to make it fail, to take the screen shot.)

The little yellow arrow is the debugger's current point in the code. The program is still running. I can use the debugger to examine variables, and execute further statements.

The lesson here is that careful research into your environment and editor leads to pushing their usability envelope, and solving *One Button Testing* for your project. Begin a project with this research, because it will assist your project developing robust error paths. At error time, a clear message must bubble up through the components of your GUI, whatever their layering, so users can respond helpfully. Nobody likes the average emergency error message—a mysterious

statement followed by program failure. Researching *Fault Navigation* for your test cases drives your research into fault navigation for your application.

This book spends time on C++ and COM because among popular GUI Toolkits they have the steepest learning curve. Their rules and infrastructure pervade today's more "advanced" Toolkit wrappers. We start by exceeding their delivered facilities. The C library `assert()` macro tunes for unexpected failures. C library vendors value efficient code when not failing.

> We value efficient programmer response at failure time.

`assert()` announces failure by invoking `__asm { int 3 }`. The extension keyword `__asm` introduces raw x86 opcodes into an executable, and `int 3` is the debugger breakpoint interrupt. But because efficient code calls that, the debugger lands deep inside the C library. I'll change a line and see:

```
SetDlgItemText(IDC_EDIT_FIRST_NAME, "Ignotz");
```

Now the test fails, and the debugger captures a hard error: "`Unhandled exception at 0x0041d652 in Project.exe: User breakpoint.`" Then it helpfully navigates to a line inside `ctr0msg.c`; that line contains `_CrtDbgBreak()`. x86 platforms expand that into `__asm { int 3 }`. Just before break-pointing, `assert()` prints this diagnostic to the console:

```
Assertion failed: "Ignatz" == name, file
                          c:\...\project\project.cpp, line 39
```

The debugger's behavior makes navigating back to the line causing the error more difficult. And the output statement reflects the variables' names, not their values. If one weren't a string literal, we would not even see `"Ignatz"`.

We need a diagnostic that:

- Trades lean code for fast fault navigation
- Streams both arguments' expression and value: `name("Ignotz")`
- Prints into both the console and the editor's Output panel
- Uses the format *`path\file.cpp(99) : message`*.

The first requirement implies our assertion macro will make our output binary much bigger than `assert()` does. It was programmer-hostile because very large programs might need to use it without getting larger. We don't care, because developer tests run in small suites, one per module.

The last two requirements permit the Visual Studio <F8> keystroke, "Go To Next Error Tag", to navigate the editor. Our diagnostics will use these techniques:

- The C++ preprocessor's stringerizer operator `#`
- MS Windows's `OutputDebugString()`
- The C++ preprocessor's `__FILE__` and `__LINE__` macros.

Putting them all together creates this:

```
#include <iostream>
#include <sstream>
```

```
        using std::cout;
        using std::endl;
        using std::stringstream;

    #define CPPUNIT_ASSERT_EQUAL(sample, result)          \
        if ((sample) != (result))  {  stringstream out;   \
            out << __FILE__ << "(" << __LINE__ << ") : "; \
            out << #sample << "(" << (sample) << ") != "; \
            out << #result << "(" << (result) << ")";     \
            cout << out.str() << endl;                    \
            OutputDebugStringA(out.str().c_str());        \
            OutputDebugStringA("\n");                     \
            __asm { int 3 }  }

        void
    test_Dialog()
    {
    …
        CPPUNIT_ASSERT_EQUAL("Ignatz", name);
…
    }
```

The stringerizer, `#`, converts an expression into a string, and `operator<<` formats the expression's value as a string. The macro inserts both these strings into a `stringstream` object. Both `cout` and `OutputDebugStringA()` reuse this object's value.

Note our macro has a common problem—it evaluates its arguments twice. Don't pass in expressions with side effects, such as `++Z`.

The macro uses `endl` so the line flushes to the console. All diagnostics must flush frequently. If the program crashes, we won't strand the last diagnostic in local memory, un-flushed. It might be trying to tell us why the program crashes.

We need the `CPPUNIT_` prefix because you should not use my introductory macros—you should understand them. You should instead use Mike Feather's CppUnit test rig, at `http://cppunit.sourceforge.net/`.

The result, at test failure time, is this line:

```
c:\…\project.cpp(56) : "Ignatz"(Ignatz) != name(Ignotz)
```

And <F8> takes us directly to the failing test assert statement.

On a platform descended from the Intel x86 architecture, if you run these tests from a command line, not the editor, the OS will treat `__asm { int 3 }` as a hard error, and it will offer to raise the default debugger. This is a mixed blessing. If a programmer attends the command line test run, they want the editor to take them to the failing line, so they can fix it. But when the command line runs in a batch, unattended, you might think you need to configure the tests to log errors instead of crash.

The two fixes here are to use a full-featured test rig, such as CppUnit, and to not have errors.

MSXML and COM (OLE, ActiveX, etc.)

Now that we have played like a cat with our mouse, we … help it pass the tests again.

```
SetDlgItemText(IDC_EDIT_FIRST_NAME, "Ignatz");
```

But the implementation still lies. A few more tests (not shown, or written) can force `OnInitDialog()` to use the data from the XML input:

```
#import <msxml4.dll>

    using MSXML2::IXMLDOMDocument2Ptr;
    typedef MSXML2::IXMLDOMNodePtr XMLNodePtr_t;

    LRESULT
ProjectDlg::OnInitDialog(UINT, WPARAM, LPARAM, BOOL &)
{
    CoInitialize(NULL);

    {

    IXMLDOMDocument2Ptr pXML;
    pXML.CreateInstance(L"Msxml2.DOMDocument.4.0");
      pXML->setProperty("SelectionLanguage", "XPath");
    pXML->async = true;
    pXML->loadXML(_bstr_t(m_xml));

    XMLNodePtr_t pName =  //  XPath strikes again!
        pXML->selectSingleNode("/user/name/text()");

    _bstr_t name = pName->text;
    SetDlgItemText(IDC_EDIT_FIRST_NAME, name);

    }  //  block to destroy the smart pointers before this:

    CoUninitialize();
    return 0;
}
```

Where did all that stuff come from?

That is near the minimum code possible to command the MSXML4 COM server to parse our XML and yield "`Ignatz`". The new lines violate our Sane Subset for COM under VC++, so the Simplicity Principles require us to upgrade them before any other development—including test-first. While we upgrade, I will explain each part.

Type Libraries and #import

Traditional libraries come with a binary library file, and an `.h` file to declare all the symbols inside it. COM objects store their own declarations inside themselves, in a special resource segment called a "type library". Separate `.h` files would be redundant. To prevent duplication, Microsoft invented a preprocessor command.

`#import <msxml4.dll>` generates and compiles two files, `msxml4.tlh` and `msxml4.tli`, that declare and define, respectively, smart wrappers on all type library symbols found in `msxml4.dll`'s resource segment.

The wrappers are smart because they use C++ objects through the header `<comdef.h>`, such as `_bstr_t` and `_variant_t`., that hide some pernicious low-level objects, such as `BSTR` and `VARIANT`, whose redundant mechanics clutter traditional COM source code. And the wrappers declare everything inside `namespace MSXML2`, to reduce the chances of collisions with similar identifiers from other libraries.

`IXMLDOMDocument2Ptr` is a smart pointer to a COM object. It will `Release()` the object automatically when it goes out of scope. These smart wrappers also hide much redundant mechanics.

`CoInitialize()` plugs our process into the COM subsystem. Without it, everything else will fail. It must call before any other function, so we so we move it to a wrapper:

```
    struct
AlohaCOM
{
    AlohaCOM()  {  CoInitialize(NULL);  }
   ~AlohaCOM()  {  CoUninitialize();    }
} g_AlohaCOM;
```

The global object `g_AlohaCOM`'s constructor calls before `main()`, and its destructor calls afterwards. Clever destructors are not cutesy; they provide for very robust code. Look up "Resource Acquisition Is Initialization" on the Internet. Our smart pointers are smart due to this technique, too—but remember they must destroy before `main()` returns and this object says "aloha" to COM the second time. So don't define any smart pointers in global storage.

```
IXMLDOMDocument2Ptr pXML;
pXML.CreateInstance(L"Msxml2.DOMDocument.4.0");
```

Those lines declare a smart pointer, and then wrap the COM utility `CoCreateInstance()`. This is a dynamic form of the Abstract Factory Pattern—it creates the target object and stores a pointer to one of its interfaces in `pXML`.

COM Error Handling

But that call to `CoCreateInstance()` might return the dreaded `HRESULT` value other than zero `0`.

Some programmers just ignore this value. Some programmers collect this value into a variable…

```
HRESULT hr = CoCreateInstance(…);
```

…and *then* ignore the value.

Here is an example of the next level of error handling (from an anonymous project):

```
void CMyDlg::OnNewRect()
{
    if (m_ctrl.AddRect(NULL, NULL, TRUE) >= 0)
    {
        // refresh UI
        EnableOK();
        UpdateToolbar();
    }
    else
     MessageBox("Sorry, a new rectangle could not be added.",
                                                  "Title");
}
```

That code should call `GetLastError()`, convert the error code to a localized string, and append this error message to the friendly apology. If functions before `AddRect()` changed state, we should change it back before displaying the error message. That prevents programs from

entering incomplete states. At error time, a program's state should roll back to the situation before the most recent user input event.

Our COM Sane Subset requires us to defend every call, and intercept any possible error. We guard our code from any problem, catch errors as early as possible, draw programmer attention directly to the lines suffering the errors, and explain the errors in English or the local equivalent. These rules help projects progress toward preventing those errors, so we must invest time before starting a project to provide the highest quality error handlers:

```
    #include <iomanip>

        using std::ostream;

        inline ostream &
    operator<< (ostream &o, _bstr_t const &str)
    {
        if (str.length())  o << str.operator const char *();
        return o;
    }

        void
    guard_( _com_error   e,
            char const * statement = NULL,
            char const * file      = NULL,
            int          line      = 0 )
    {
        stringstream out;

        if (file)
            out << file << "(" << line << ") : ";

        out << "COM error 0x";
        out.fill('0');
        out << std::setw(8) << std::hex << e.Error();

        if (statement)
            out << " executing: " << statement;

        out << "\n\t" << e.Description();
        out << "\n\t" << e.ErrorMessage();
        cout << out.str() << endl;
        OutputDebugStringA(out.str().c_str());
        OutputDebugStringA("\n");
    }

    #ifdef _DEBUG
    #   define ELIDE_(x) x
    #else
    #   define ELIDE_(x) // x
    #endif

    #define GUARD(statement) do { HRESULT hr = (statement);  \
        if (FAILED(hr))  {                                   \
            guard_(hr, #statement, __FILE__, __LINE__);      \
            ELIDE_(__asm { int 3 });                         \
            _com_issue_error(hr); } } while(false)
…
        int
    main()
    {
```

```
        try {
            test_Dialog();
            }
        catch(_com_error &e)
            {
            guard_(e);
            ELIDE_(__asm { int 3 });
            return 1;
            }
        return 0;
    }
```

Briefly, that system fulfills these requirements at failure time:

- <F8> navigates to the statement that failed
  - `__FILE__, __LINE__`
- when compiled in `_DEBUG` mode, the debugger break-points on the line that failed
  - `ELIDE_(__asm { int 3 });`
- when compiled in Release mode, the guarded error only throws an exception
  - `_com_issue_error(hr);`
- the COM error displays with its number in hex format
  - `out << std::setw(8) << std::hex << e.Error();`
- the COM error displays with its text error messages
  - `_com_error e (hr);`

```
out << "\n\t" << e.Description();
out << "\n\t" << e.ErrorMessage();
```

- the error messages display to both the console and the editor's Output panel
  - `stringstream` (which we have seen before)
- the entire `GUARD()` statement must have a healthy `;` on the end
  - `do…while(false)`
- after reporting the error's local details, we `throw` the error
  - `_com_issue_error(hr)`.

Something must `catch` those `thrown` errors. A real GUI error handler would return the program to the state where it was before the user's last input, and would display a localized error message in an appropriate on-screen format. But our new lines in `main()` merely tell the calling environment that the tests failed. More advanced COM-enabled test rigs could permit the next test to run.

A note about the rampant `#define` situation. One should not `#define` small functions that could use `inline`. Our little functions need these preprocessor abilities:

- stringerization `#`
- token pasting `##`
- conditionally `ELIDE_()` non-production code
- `__FILE__` and `__LINE__` expand in the calling code
- `__asm { int 3 }` breakpoints the calling code.

Those facilities permit the most efficient diagnostics. We will return to token pasting.

To test `GUARD()` (temporarily—not permanently!), we add it to a statement that needs it, then we screw that statement up:

```
    GUARD(pXML.CreateInstance(L"Msxml2.DOMDocument.6.0"));
```

Until MS unveils the sixth version of `DOMDocument`, that line will produce a diagnostic like this:

```
c:\...\project.cpp(101) : COM error 0x800401f3 executing:
            pXML.CreateInstance(L"Msxml2.DOMDocument.6.0")

    Invalid class string
```

Every line that deals in COM could have a failing `HRESULT` somewhere inside. Not every such line has a `GUARD()`, because statements that declare a "return value" in their type library interface definitions receive normal C++ return values when `#import` wraps them. The low-level COM system passes the address of the declared return value in, and returns an `HRESULT`, which the wrapper itself checks.

Here's a sample wrapper. I added `this->` for clarity:

```
inline VARIANT_BOOL IXMLDOMDocument::loadXML (_bstr_t bstrXML) {
    VARIANT_BOOL _result = 0;
    HRESULT _hr = this->raw_loadXML(bstrXML, &_result);
    if (FAILED(_hr)) _com_issue_errorex(_hr, this, __uuidof(this));
    return _result;
}
```

`raw_loadXML()` is a C++ method inside the target object. COM forced `*this` to union with that object in memory, using implementation-specific techniques. So the `raw_loadXML()` call is down to within one indirection of the actual hardware. The return value, `_result`, secretly passes by address as the last argument. The wrapper then tests the raw return value, `_hr`, and throws an exception if it fails.

If these kinds of expressions fail, we must catch their exceptions, then run the debugger to find the line that blew. If this becomes an issue, we could write a new version of `GUARD()` that, in Debug Mode, wraps each call with `try`, `catch`, and `guard_()`. And VC++ may have options to breakpoint `throw` statements.

Test rigs can often recover from unexpected exceptions in one test case, allowing others to run. In a few pages we will write a TestCase system that could easily support a `catch(...)` statement, for projects that need it. But we must fix any low-level error as soon as we see it—we rarely need to see a list of them.

Contrarily, softer languages automate this *Fault Navigation* into their editors—often in ways that deny us complete control. With a harder language, and the preprocessor, we have more control over fault recovery.

XML Syntax Errors

These errors fortunately appear on a channel different from generic COM errors. Here's a quick, simple way to make them useful:

```
    inline _bstr_t
formatParseError( IXMLDOMDocument2Ptr &pXML )
{
```

```
        MSXML2::IXMLDOMParseErrorPtr pError = pXML->parseError;

        return _bstr_t("At line ") +
               _bstr_t(pError->Getline()) +
               _bstr_t("\n") +
               _bstr_t(pError->Getreason());
    }
...
        VARIANT_BOOL parsedOkay = pXML->loadXML(_bstr_t(m_xml));

        if (!parsedOkay)
            {
            cout << formatParseError(pXML);
            ELIDE_(__asm { int 3 });
            return 0;
            }
```

If the XML fails to parse, we print a few error details to the console, then—in Debug mode—we call a breakpoint that raises the debugger. The Release Mode version could reuse `formatParseError()` to report these details back to whoever allegedly wrote faulty XML.

That is enough error handling for now. Those with experience debugging poorly written COM code, with no error handling, will appreciate how easily we can now write new statements, and how aggressively our entire system constrains them.

We bootstrapped a dialog, a tiny test rig, and an XML parser. I will print out the entire current source, without comment, and then go to the next feature:

```
#include <atlStr.h>
#include <atlApp.h>

    CAppModule _Module;

#include <atlWin.h>
#include <atlDlgs.h>
#include "resource.h"
#include <iostream>
#include <sstream>
#include <iomanip>
#import <msxml4.dll>

    using MSXML2::IXMLDOMDocument2Ptr;
    typedef MSXML2::IXMLDOMNodePtr XMLNodePtr_t;

    using std::cout;
    using std::endl;
    using std::ostream;
    using std::stringstream;

    struct
AlohaCOM
{
    AlohaCOM()  {  CoInitialize(NULL);  }
   ~AlohaCOM()  {  CoUninitialize();    }
} g_AlohaCOM;

    inline ostream &
operator<< (ostream &o, _bstr_t const &str)
{
    if (str.length())  o << str.operator const char *();
    return o;
}
```

```
    void
guard_( _com_error   e,
        char const * statement = NULL,
        char const * file      = NULL,
        int          line      = 0 )
{
    stringstream out;

    if (file)
        out << file << "(" << line << ") : ";

    out << "COM error 0x";
    out.fill('0');
    out << std::setw(8) << std::hex << e.Error();

    if (statement)
        out << " executing: " << statement;

    out << "\n\t" << e.Description();
    out << "\n\t" << e.ErrorMessage();
    cout << out.str() << endl;
    OutputDebugStringA(out.str().c_str());
    OutputDebugStringA("\n");
}

#ifdef _DEBUG
#   define ELIDE_(x) x
#else
#   define ELIDE_(x) // x
#endif

#define GUARD(statement) do { HRESULT hr = (statement);  \
    if (FAILED(hr))  {                                   \
        guard_(hr, #statement, __FILE__, __LINE__);      \
        ELIDE_(__asm { int 3 });                         \
        _com_issue_error(hr); } } while(false)

#define CPPUNIT_ASSERT_EQUAL(sample, result)          \
    if ((sample) != (result))  {  stringstream out;   \
        out << __FILE__ << "(" << __LINE__ << ") : "; \
        out << #sample << "(" << (sample) << ") != "; \
        out << #result << "(" << (result) << ")";     \
        cout << out.str() << endl;                    \
        OutputDebugStringA(out.str().c_str());        \
        OutputDebugStringA("\n");                     \
        __asm { int 3 }  }

    class
ProjectDlg:
    public CDialogImpl<ProjectDlg>
{
  public:
    enum { IDD = IDD_PROJECT };

    BEGIN_MSG_MAP(ProjectDlg)
        MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
    END_MSG_MAP()

    LRESULT OnInitDialog(UINT, WPARAM, LPARAM, BOOL &);
    ProjectDlg(char const *xml): m_xml(xml) {}
    CString getText(UINT ID) const;
```

```
  private:
    CString m_xml;
};

    inline _bstr_t
formatParseError( IXMLDOMDocument2Ptr &pXML )
{
    MSXML2::IXMLDOMParseErrorPtr pError = pXML->parseError;

    return _bstr_t("At line ") +
           _bstr_t(pError->Getline()) +
           _bstr_t("\n") +
           _bstr_t(pError->Getreason());
}

    LRESULT
ProjectDlg::OnInitDialog(UINT, WPARAM, LPARAM, BOOL &)
{
    IXMLDOMDocument2Ptr pXML;
    GUARD(pXML.CreateInstance(L"Msxml2.DOMDocument.4.0"));
    pXML->setProperty("SelectionLanguage", "XPath");
    pXML->async = true;

    VARIANT_BOOL parsedOkay = pXML->loadXML(_bstr_t(m_xml));

    if (!parsedOkay)
        {
        cout << formatParseError(pXML);
        ELIDE_(__asm { int 3 });
        return 0;
        }

    XMLNodePtr_t pName =
        pXML->selectSingleNode("/user/first_name/text()");

    _bstr_t name = pName->text;
    SetDlgItemText(IDC_EDIT_FIRST_NAME, name);
    return 0;
}

    void
test_Dialog()
{
        char const *
    xml = "<user>"
            "<first_name>Ignatz</first_name>"
            "<last_name>Mouse</last_name>"
            "</user>";

    ProjectDlg aDlg(xml);
    aDlg.Create(HWND_TOP);
    CString name;
    aDlg.GetDlgItemText(IDC_EDIT_FIRST_NAME, name);
    CPPUNIT_ASSERT_EQUAL("Ignatz", name);
    aDlg.DestroyWindow();
}

    int
main()
{
    try {
        test_Dialog();
```

```
        }
    catch(_com_error &e)
        {
        guard_(e);
        ELIDE_(__asm { int 3 });
        return 1;
        }
    return 0;
}
```

Proceed to the Next Ability

C++ requires this level of complexity because it is both a high-level language and "Portable Assembler". The error handling code must take care of details as high as language-agnostic COM objects dynamically bonded to our application (`pXML->parseError`), and as low as individual opcodes (`__asm { int 3 }`). Large, performance-bound systems that must solve these problems, consistently and coherently, have a reputation for unreliability.

The emerging test rig now constrains three separate features—COM, test cases, and dialogs. C++ forces programmers to take responsibility for plumbing that softer languages hide. Our small, low-performance application could scale into a huge, reliable system. Ensuring every error gets noticed, instantly, converts responsibility into control.

Everything gets much simpler, very soon. First we use test-first to force the dialog to provide some easy behaviors, then we respond to more duplication by finishing the test rig. From then on, developing this project will rapidly re-use all these features, following much the same path as a higher-level system would provide.

A `first_name` requires a `last_name`. Clone and modify a test:

```
    void
test_Dialog_lastName()
{
        char const *
    xml = "<user>"
            "<first_name>Ignatz</first_name>"
            "<last_name>Mouse</last_name>"
            "</user>";

    ProjectDlg aDlg(xml);
    aDlg.Create(HWND_TOP);
    CString name;
    aDlg.GetDlgItemText(IDC_EDIT_LAST_NAME, name);
    CPPUNIT_ASSERT_EQUAL("Mouse", name);
    aDlg.DestroyWindow();
}
```

Clone and modify some resources so the test compiles and fails for the correct reason:

```
#define IDC_EDIT_FIRST_NAME           1001
#define IDC_EDIT_LAST_NAME            1002
…
    EDITTEXT        IDC_EDIT_FIRST_NAME,
                        61,17,53,13,ES_AUTOHSCROLL
    LTEXT           "Last Name",IDC_STATIC,7,28,45,10
    EDITTEXT        IDC_EDIT_LAST_NAME,
                        61,27,53,13,ES_AUTOHSCROLL
```

And clone and modify the implementation so the tests pass:

```
    XMLNodePtr_t pFirstName =
        pXML->selectSingleNode("/user/first_name/text()");

    _bstr_t first_name = pFirstName->text;
    SetDlgItemText(IDC_EDIT_FIRST_NAME, first_name);

    XMLNodePtr_t pLastName =
        pXML->selectSingleNode("/user/last_name/text()");

    _bstr_t last_name = pLastName->text;
    SetDlgItemText(IDC_EDIT_LAST_NAME, last_name);
```

As usual, passing a test makes the design worse. Both the test cases and production code need improvements, but we will improve the tests first.

The code that retrieves a string from an edit field is redundant, and easy to merge:

```
    CString
ProjectDlg::getText(UINT ID) const
{
    CString text;
    GetDlgItemText(ID, text);
    return text;
}

        char const *
    xml = "<user>"
            "<first_name>Ignatz</first_name>"
            "<last_name>Mouse</last_name>"
            "</user>";

    void
test_Dialog()
{
    ProjectDlg aDlg(xml);
    aDlg.Create(HWND_TOP);

    CPPUNIT_ASSERT_EQUAL( "Ignatz",
                aDlg.getText(IDC_EDIT_FIRST_NAME) );

    aDlg.DestroyWindow();
}

    void
test_Dialog_lastName()
{
    ProjectDlg aDlg(xml);
    aDlg.Create(HWND_TOP);

    CPPUNIT_ASSERT_EQUAL( "Mouse",
                aDlg.getText(IDC_EDIT_LAST_NAME) );

    aDlg.DestroyWindow();
}
```

There's a small victory. The new method `getText()` returns a `CString`, so the tests have fewer lines.

The remaining lines, however, are mostly the same. The tests need to share a class with a member `m_aDlg`, so the class can construct and destruct it stereotypically, while each test focuses on its one assertion. That requires a class for `m_aDlg` to become a member of:

```
    class
TestCase
{
  public:
    virtual void setUp() {}
    virtual void runCase() = 0;
    virtual void tearDown() {}
};

    class
TestDialog:  public TestCase
{
  public:
    ProjectDlg m_aDlg;

    virtual void setUp()
                    { m_aDlg.Create(HWND_TOP); }

    virtual void tearDown()
                    { m_aDlg.DestroyWindow(); }

    TestDialog():  m_aDlg(xml)  {}
};

    class
TestDialog_first_name:  public TestDialog
{
  public:
    void
runCase()
  {
    CPPUNIT_ASSERT_EQUAL( "Ignatz",
        m_aDlg.getText(IDC_EDIT_FIRST_NAME) );
  }
};

    class
TestDialog_last_name:  public TestDialog
{
  public:
    void
runCase()
  {
    CPPUNIT_ASSERT_EQUAL( "Mouse",
        m_aDlg.getText(IDC_EDIT_LAST_NAME) );
  }
};

    int
main()
{
    try {
        TestDialog_first_name test1;
        test1.setUp();
        test1.runCase();
```

```
        test1.tearDown();

        TestDialog_last_name test2;
        test2.setUp();
        test2.runCase();
        test2.tearDown();
        }
    catch(_com_error &e)
        {
        guard_(e);
        ELIDE_(__asm { int 3 });
        return 1;
        }
    return 0;
}
```

What's going on? We thought that duplication removal would make a better design. That design has more duplication!

Test Collector

That was the first phase of "Grind it 'Till you Find it". We built two structures that look very similar. Next, enhance `TestCase`, to remove duplication from `main()`:

```
#include <list>

    class
TestCase
{
  public:
    typedef std::list<TestCase *> TestCases_t;
    TestCases_t static cases;

    TestCase()  {  cases.push_back(this);  }
    virtual void setUp() {}
    virtual void runCase() = 0;
    virtual void tearDown() {}
};

TestCase::TestCases_t TestCase::cases;

…
    class
TestDialog_last_name:  public TestDialog
{
  public:
    void
runCase()
  {
    CPPUNIT_ASSERT_EQUAL( "Mouse",
        m_aDlg.getText(IDC_EDIT_LAST_NAME) );
  }
};
TestDialog_last_name test2;

    int
main()
{
    try {
    TestCase::TestCases_t::iterator it (TestCase::cases.begin());

        for ( ;  it != TestCase::cases.end();  ++it )
```

```
            {
            TestCase &aCase = **it;
            aCase.setUp();
            aCase.runCase();
            aCase.tearDown();
            }
        }
```

The classes `TestDialog_first_name` (not shown) and `TestDialog_last_name` now look very similar. Ideally, the mechanics of each test case don't need to repeat. Adding new cases will be very easy.

To merge the redundant structure, write a macro that declares a class and instantiate an object in one command:

```
#define TEST_(suite, target)            \
    struct suite##target:  public suite \
    { void runCase(); }                 \
    a##suite##target;                   \
    void suite##target::runCase()

TEST_(TestDialog, first_name)
{
    CPPUNIT_ASSERT_EQUAL( "Ignatz",
        m_aDlg.getText(IDC_EDIT_FIRST_NAME) );
}

TEST_(TestDialog, last_name)
{
    CPPUNIT_ASSERT_EQUAL( "Mouse",
        m_aDlg.getText(IDC_EDIT_LAST_NAME) );
}
```

New cases are now easy. Fixtures and Test Isolation are very easy. Classes like `TestDialog` take care of setting up and tearing down tested objects, providing a framework to share other common operations on the tested objects.

The Test Collector pattern ensures when we write new cases, we can't forget to add them to a remote list of cases. The `TEST_()` macro builds `aTestDialogfirst_name`, etc, and plugs it into the polymorphic array, `TestCase::cases`.

Every unique case provides a specialized `runTest()` method, and reuses its base class's member variables. A "Test Suite" is "that set of test cases which share the same Fixtures", beginning with the standard fixtures `setUp()` and `tearDown()`.

But these `TEST_()` macros don't look like normal C++ functions!

C++ Heresy

C++ requires a hyperactive Sane Subset. Many successful C++ projects obey too many guidelines (and some succeed despite obeying too few). To get things done, this project follows some uncommon guidelines. They need explanations to link them to the common style guidelines that C++ programmers learned.

Registered Package Suffixes

Only C++ implementers may put an underbar _ on the front of most kinds of identifiers. I put test macros into a pretend namespace, with _ on the ends of short identifiers, because nobody else does. Similarly, Microsoft claims all classes beginning with `C` to be their territory.

That is reason enough not to trespass; programmers using Microsoft libraries should *not* write new classes with the `C` prefix.

See the book *Large Scale C++ Software Design*, by John Lakos, for construction and testing tips that work at all scales, including an alternative to `namespaces` called "Registered Package Prefixes". Macros can't use `namespaces`, so they need the manual alternative. My test macros avoid identifier collisions with any other prefixes anywhere by using _ as a registered package suffix.

Warts

This program has agreed on a common abbreviation for its most-used term, "dialog". Engineers reading this code will learn to pronounce `Dlg` as "dialog".

Decorating identifiers with warts (an application-specific "Hungarian Notation") provides rapid visual feedback regarding identifiers' intents. English introduces objects with "a", so `aDlg` is pronounced "a dialog".

To merge our styles with our library, we prefix (most) member variables with `m_`. So `m_aDlg` is pronounced either "a dialog member" (or maybe "mad-dialog").

Better names, in retrospect, were possible. Blame my muscle memory.

Macros

The `TEST_()` macro must create a uniquely named class, and a single object, to distinguish its `runCase()` method. The macro uses token pasting, `##`, to paste its parent name to the `target` in the macro's second argument. This documents the name of the primary class, method, or ability under test. The macro creates a global instance of the resulting object, with a long but unique name. When this global object constructs, it enrolls itself in the global list of all test cases.

Test Insulation

Many "assert" systems don't permit their program to continue after a fault. The assertions we wrote, however, permit the Debug Step commands to resume execution. (Tests are an excellent platform to run debuggers to "step through" ones code—just remember tests must come first!)

If a program must stop before its behavior becomes undefined, use a generic assertion, such as assert(), or a library specific `ATLASSERT()`.

The *Unit test rigs often provide assertions that count the number of successes and failures. Some raise a list box with all the results of failing tests. Those features should here be easy to add, if you think you need them.

Turbulence

Flow is a very important goal of programming. However, I'm using the 2002 version of DevEnv. When I edit files and hit DevEnv's "Go" button, it asks a stupid question. It raises a dialog listing all the changed source files, and asks, "…Would you like to build them?"

99.99% of the time any engineer knows what's in the list, won't read it, and will bang the <Spacebar> to answer "Yes" (or their locale's equivalent). Visual Studio 6 asked the same question, but you could reliably hit <F5><Spacebar> rapidly, and the <Spacebar> buffered until the dialog began to display and pulled it from the event queue. Due to VS7's advanced and decoupled architecture, you must wait for the dialog to display before tapping <Spacebar> to dismiss it. This violates Microsoft's own usability guidelines.

Then, to further inhibit Flow, when you start a compile, DevEnv inexplicably refocuses the keyboard into the Output panel. You must then bang <Escape> to get it out. Newer releases might address these usability issues.

At least this editor responds better than Visual Basic 6. That editor's factory default configuration, when you execute a program, does not save all your files. The editor couples with your program at runtime, so if your program crashes, you can lose all changes. The editor permits changing the source code around a breakpoint, but not saving your change. If you program in the debugger, then crash, you will lose these changes too.

Don't "Edit and Continue"

Visual C++, to compete with Visual Basic, permits a partial recompile of code changes while debugging. If you set a breakpoint, run, stop at the breakpoint, edit the code, and run again, VC++ will recompile parts of a live program before running statements after the breakpoint.

Turn this feature off. The only reasons to edit at a breakpoint are adding a comment, adding an assertion, or changing the code. Changing code requires a fresh test. The "Edit and Continue" feature supports programmers without Test Isolation, who must navigate their programs for a long time before reaching a breakpoint. Incremental testing rarely has that problem.

No test run should cover two phases of the same code. After changing code, always restart your tests. If you hesitate, your tests are insufficiently incremental.

Don't Typecast

My C++ Sane Subset puts typecasts lower than explicit `operator` calls. That explains the `.operator char const *()` statements. Explicit `operator` calls are less liable to break if their left operand changes type. In a Static Typing language, a type check is a test. Don't suppress these tests with typecasts, and don't learn from Microsoft's sample code, and typecast everything to what it already is.

Sometimes legacy systems force a typecast. (One appears on page 236). Do not write a C-style cast, such as `(unsigned char*)"x"`. Use an elaborate, named cast, such as `static_cast<unsigned char const *>("yo")`. Among their other benefits, they can enforce `const` correctness.

Const Correctness

The C++ keyword `const` instructs compilers to reject overt attempts to change a variable's value. Covert attempts produce undefined behavior, meaning anything could happen. Covert attempts are possible because C languages use weak typing to compete with typeless Assembler.

C++ functions can take arguments by copy, by address, or by reference. Ideally, if an object passed into a function does not change, the object should pass by copy:

```
void foo(SimCity aCity);
```

That code is inefficient. In general, programmers should not stress about efficiency until they have enough code to measure it and find the slow spots. In this situation, a more efficient implementation has an equal mental cost. When we pass by reference, our program spends no time making a huge copy of an entire city:

```
void foo(SimCity &aCity);
```

Now if `foo()` won't change that city's value, the function should declare that intention in its interface, using pass-by-constant-reference to simulate pass-by-copy:

```
void foo(SimCity const &aCity);
```

That call syntax is both cognitively and physically efficient. It's cognitively efficient because it gives `foo()` no copied object to foolishly change and then discard. Statements inside `foo()` that might attempt to change that city shouldn't compile. They should create syntax errors that give developers early feedback. It's physically efficient because the compiler produces opcodes that only give `foo()` a handle to an existing city, without copying it.

C++ supports qualifications before their qualified types, such as "`const SimCity &`". That is an exception to the rule that `const` modifies the type on its left. Expressions should start with their most important type, if possible, so I write `SimCity const &` with the `const` after the type it qualifies.

Latent Modules

I did not yet split all this source into four modules. They could be, for example, `test.cpp`, `dialog.cpp`, `xml.cpp` and `dialog_test.cpp`. Refactoring can lead to the benefits of decoupling without separate files. A method obeys the *Clear and Expressive* Simplicity Principle when it only targets one library, not two or three. Refactoring to build short methods, then pulling them together by affinity, creates classes that will need no further changes when the time comes to move them into their own modules.

Also, the `class` definitions are not yet in `.h` files. Don't write them there by default; only move a class into a `.h` file if other `.cpp` files need to see it. All identifiers should have the narrowest scope possible, so classes should be inside `.cpp` files if possible.

Use the Console in Debug Mode

When `main()` compiles in Release mode, conditional compilation should censor all those pesky tests, and should raise our dialog. (Or another application could reuse the dialog, so our module cannot run alone without tests.)

If our own project raises our dialog, then in Release mode we should switch the linker to use `/SUBSYSTEM:WINDOWS`, and should set the entry-point symbol to something like "`_tWinMain`". Create a scratch Windows project and examine its compiler settings to learn how to switch your project over. Alternately, create a new project in Windows mode, and copy your Debug settings into that.

Enabling

This chapter contains many fine tips regarding the intersection of C++, COM, Visual Studio, and WTL. Some Windows SDK programs are crufty and bloated. This Case Study's source code is clean, by contrast, partly because it's carefully refactored, and partly because it only uses simple SDK techniques.

If you compile this chapter's source, to follow along & practice, and if you depart from the script toward a different WTL or SDK feature, you will find yourself in a curious situation. GUI Toolkits come with documentation that overwhelmingly assumes readers used that toolkit's wizards and form painters. The documentation expects readers to paint controls and

manually test them, not treat toolkits like libraries, query their state, simulate their users, or abuse their event queues.

The best programming documentation site, `http://groups.google.com/`, contains hundreds of thousands of excellent questions and answers (and the occasional colorful flame;). The programming questions usually request help deviating from the outputs of wizards. For the near future, if you search or ask a question that readers of *TFUI* would consider obvious, such as, "How to I write an Internet Explorer plug-in so that an HTML page containing JavaScript can simulate user input and test that plug-in's behavior?" you will receive the electronic equivalents of funny looks.

Until the online archives of conversations between Agile engineers improve this situation, the best advice to deviate from this chapter's WTL code is to not deviate too far, to start a project again with more standard infrastructure, and to occasionally consult those wizards for advice.

All Tests Passed

Meanwhile, our project's `main()` has no useful contents yet. It can't do anything more than call the tests. `ProjectDlg` has a simple interface, so writing the production version of `main()` will be easy.

The test version of `main()`, after running tests, must return `0` if they all pass, or `1` if any fail. A top-level script can call this program and others in batches, then accurately report "`All tests passed`" at the end of an integration test run, if and only if every test in every module passed.

Every time you assemble a test rig, you must remember this rule: Collect the test runner's result, and pass it into `exit()`, or `return` it from `main()`. A test runner can't call your `exit()` for you, so you must remember to configure each test program's `main()` correctly.

To report failure, `CPPUNIT_ASSERT_EQUAL()`, at failure time, will assign `false` to `all_tests_passed`, a member of TestCase. This new feature makes the macro `CPPUNIT_-ASSERT_EQUAL()` into a *de-facto* member of `TestCase`:

```
    #define CPPUNIT_ASSERT_EQUAL(sample, result)          \
        if ((sample) != (result))  {  stringstream out;   \
…
            all_tests_passed = false;                    \
            __asm { int 3 }  }

        class
    TestCase
    {
…
        static bool runTests();
      protected:
        static bool all_tests_passed;
    };

    bool TestCase::all_tests_passed = true;
    TestCase::TestCases_t TestCase::cases;
```

```
    bool
TestCase::runTests()
{
    TestCase::TestCases_t::iterator it (TestCase::cases.begin());

    for ( ;  it != TestCase::cases.end();  ++it )
        {
        TestCase &aCase = **it;
        aCase.setUp();
        aCase.runCase();
        aCase.tearDown();
        }
    return TestCase::all_tests_passed;
}
```

Moving those private details into `TestCase`'s implementation permits a shorter `main()`:

```
    int
main()
{
    try {
        return !TestCase::runTests();
        }
…
    }
```

Keep Tests Easy to Write

In this project, so far, we created a dialog, without a wizard, and without a test rig. Writing two tests, to create two edit fields (first and last name) provided duplication. Merging it created a test rig.

The next feature is the "Address 1:" field. To create its test, we add the `<address_1>` tag to the global `xml` resource:

```
        char const *
    xml = "<user>"
            "<first_name>Ignatz</first_name>"
            "<last_name>Mouse</last_name>"
            "<address_1>Heppy Land</address_1>"
            "</user>";
…
TEST_(TestDialog, address_1)
{
    CPPUNIT_ASSERT_EQUAL( "Heppy Land",
        m_aDlg.getText(IDC_EDIT_ADDRESS_1) );
}
```

That test doesn't compile, because our resources need more edit fields. We'll finish authoring the dialog now so we won't need to come back to it:

```
#define IDD_PROJECT                     101
#define IDC_EDIT_FIRST_NAME            1001
#define IDC_EDIT_LAST_NAME             1002
#define IDC_EDIT_ADDRESS_1             1003
#define IDC_EDIT_ADDRESS_2             1004
#define IDC_EDIT_CITY                  1005
#define IDC_EDIT_STATE                 1006
#define IDC_EDIT_ZIP                   1007

...
IDD_PROJECT DIALOGEX 0, 0, 187, 94
STYLE DS_SETFONT | DS_MODALFRAME | DS_FIXEDSYS |
    WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Project"
FONT 8, "MS Shell Dlg", 400, 0, 0x1
BEGIN
    DEFPUSHBUTTON   "Okay",IDOK,130,7,50,14
    PUSHBUTTON      "Cancel",IDCANCEL,130,24,50,14
    LTEXT           "First Name",IDC_STATIC,7,18,45,10
    EDITTEXT        IDC_EDIT_FIRST_NAME,
                        61,17,53,13,ES_AUTOHSCROLL
    LTEXT           "Last Name",IDC_STATIC,7,28,45,10
    EDITTEXT        IDC_EDIT_LAST_NAME,
                        61,27,53,13,ES_AUTOHSCROLL
    LTEXT           "Address 1",IDC_STATIC,7,38,45,10
    EDITTEXT        IDC_EDIT_ADDRESS_1,
                        61,37,53,13,ES_AUTOHSCROLL
    LTEXT           "Address 2",IDC_STATIC,7,48,45,10
    EDITTEXT        IDC_EDIT_ADDRESS_2,
                        61,47,53,13,ES_AUTOHSCROLL
    LTEXT           "City",IDC_STATIC,7,58,45,10
    EDITTEXT        IDC_EDIT_CITY,
                        61,57,53,13,ES_AUTOHSCROLL
    LTEXT           "State",IDC_STATIC,7,68,45,10
    EDITTEXT        IDC_EDIT_STATE,
                        61,67,53,13,ES_AUTOHSCROLL
    LTEXT           "ZIP",IDC_STATIC,7,78,45,10
    EDITTEXT        IDC_EDIT_ZIP,
                        61,77,53,13,ES_AUTOHSCROLL
END
```

This script compiles into a resource segment. Without it, the dialog's constructor would need dozens of calls to functions to create those controls and Set all their properties.

Now clone more code, and make that test pass:

```
    LRESULT
ProjectDlg::OnInitDialog(UINT, WPARAM, LPARAM, BOOL &)
{
    IXMLDOMDocument2Ptr pXml;
    GUARD(pXml.CreateInstance(L"Msxml2.DOMDocument.4.0"));
    pXML->setProperty("SelectionLanguage", "XPath");
    pXml->async = true;

    VARIANT_BOOL parsedOkay = pXml->loadXML(_bstr_t(m_xml));

    if (!parsedOkay)
        {
        cout << formatParseError(pXml);
        ELIDE_(__asm { int 3 });
        return 0;
        }

    XMLNodePtr_t pFirstName =
```

```
        pXML->selectSingleNode("/user/first_name/text()");

    _bstr_t first_name = pFirstName->text;
    SetDlgItemText(IDC_EDIT_FIRST_NAME, first_name);

    XMLNodePtr_t pLastName =
        pXML->selectSingleNode("/user/last_name/text()");

    _bstr_t last_name = pLastName->text;
    SetDlgItemText(IDC_EDIT_LAST_NAME, last_name);

    XMLNodePtr_t pAddress_1 =
        pXML->selectSingleNode("/user/address_1/text()");

    _bstr_t address_1 = pAddress_1->text;
    SetDlgItemText(IDC_EDIT_ADDRESS_1, address_1);
    return 0;
}
```

That function is growing some duplication. We delayed folding it together for so long (until 3 instances instead of 2!), because building the test rig took so much effort. That is now stable, so we can attend to the production code duplication. It leads to a familiar destination.

Chapter 8: *Model View Controller*

Many competitive GUI Toolkits support a "Class Wizard" to generate an application skeleton, and to add stereotypical features to them. Only use these tricksters to start a project, or as documentation. Run them in a scratch folder, just to see what they know.

If a wizard starts your project, treat its generated code like legacy code. Introduce tests to drive each new change. Until our tool vendors catch up with us, don't trust wizards to add new features. And don't hold your breath waiting for a wizard that generates the test rig with the new windows.

The last Case Study bypassed the VC++ Class Wizard, built a test runner from scratch, plugged in MSXML for a Representation Layer, and produced the beginning of a data entry form. Our dialog class still looks primitive:

This Case Study finishes its logic, and the next one will improve its esthetics. Fictitious User Stories for this iteration lead us to:

- Page 197: Enable Temporary Interactive Tests.
- 200: Fold duplicated definitions of behavior into prototypical MVC.
- 206: Add a new requirement that breaks the design.
- 209: Refactor the design into simple MVC.
- 214: Compare MVC to DDX.

The last chapter began a class, `ProjectDlg`, which lives within the Windows Template Library framework. It inherits a template expanded on its own name, following "Coplien's Curiously Recurring Template Pattern". Base classes in WTL enforce typesafety by knowing who derives from them. For example, code inside `CDialogImpl` will downcast `*this` into a `ProjectDlg` to get the message map that `BEGIN_MSG_MAP()` creates.

Those of your screaming, "That's what virtual methods are for!" are right. Always remember you don't need to do the same things WTL does just to use it. This use of the Recurring Template Pattern might have benefits, so I'll just claim they are outside this book's scope.

```
    class
ProjectDlg:
    public CDialogImpl<ProjectDlg>
{
  public:
```

```
    enum { IDD = IDD_PROJECT };

    BEGIN_MSG_MAP(ProjectDlg)
        MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
    END_MSG_MAP()

    ProjectDlg(char const *xml): m_xml(xml) {}
    LRESULT OnInitDialog(UINT, WPARAM, LPARAM, BOOL &);
    CString getText(UINT ID) const;

  private:
    CString m_xml;
};
```

The dialog class has a brief constructor, and two relevant methods. The message map calls `OnInitDialog()` as the wrapped window object constructs. The `WM_INITDIALOG` message is MS Windows' language-neutral constructor system. Its code looks smaller and cleaner than an SDK dialog box.

But its implementation of `OnInitDialog()`, back on page 192, is very long and crufty. To clean `OnInitDialog()` up, start with a naïve Extract Method Refactor. No particular goal in mind—just suppressing the most common duplication:

```
    static _bstr_t
get(IXMLDOMDocument2Ptr &pXML, _bstr_t field)
{
    XMLNodePtr_t pField =
        pXML->selectSingleNode("/user/" + field + "/text()");

    return pField->text;
}

    LRESULT
ProjectDlg::OnInitDialog(UINT, WPARAM, LPARAM, BOOL &)
{
    IXMLDOMDocument2Ptr pXML;
    GUARD(pXML.CreateInstance(L"Msxml2.DOMDocument.4.0"));
    pXML->setProperty("SelectionLanguage", "XPath");
    pXML->async = true;

    VARIANT_BOOL parsedOkay = pXML->loadXML(_bstr_t(m_xml));

    if (!parsedOkay)
        {
        cout << formatParseError(pXML);
        ELIDE_(__asm { int 3 });
        return 0;
        }

    SetDlgItemText(IDC_EDIT_FIRST_NAME, get(pXML, "first_name"));
    SetDlgItemText(IDC_EDIT_LAST_NAME,  get(pXML, "last_name"));
    SetDlgItemText(IDC_EDIT_ADDRESS_1,  get(pXML, "address_1"));
    return 0;
}
```

That looks healthier, because when the time comes to test "`address_2`" into use, I can envision the exact keystrokes needed to clone a line and modify it.

But one good architectural goal is separation of concerns. Part of our Representation Layer is the code that constructs a `DOMDocument`. The GUI Layer should not know where the data comes from, and a decoupled layer would express intent more clearly.

Perform the Extract Class Refactor around `pXML`, and move the new `get()` into the new class:

```
    class
CustomerAddress
{
  public:
    CustomerAddress(CString xml)
        {
        GUARD(m_pXML.CreateInstance(L"Msxml2.DOMDocument.4.0"));
        m_pXML->setProperty("SelectionLanguage", "XPath");
       m_pXML->async = true;

        VARIANT_BOOL parsedOkay = m_pXML->loadXML(_bstr_t(xml));

        if (!parsedOkay)
            {
            cout << formatParseError(m_pXML);
            ELIDE_(__asm { int 3 });
            }
        }
    _bstr_t get(_bstr_t field);

  private:
    IXMLDOMDocument2Ptr m_pXML;
};

    _bstr_t
CustomerAddress::get(_bstr_t field)
{
    XMLNodePtr_t pField =
     m_pXML->selectSingleNode("/user/" + field + "/text()");

    return pField->text;
}

    LRESULT
ProjectDlg::OnInitDialog(UINT, WPARAM, LPARAM, BOOL &)
{
    CustomerAddress aCA(m_xml);
    SetDlgItemText(IDC_EDIT_FIRST_NAME, aCA.get("first_name"));
    SetDlgItemText(IDC_EDIT_LAST_NAME,  aCA.get("last_name"));
    SetDlgItemText(IDC_EDIT_ADDRESS_1,  aCA.get("address_1"));
    return 0;
}
```

We have a primordial Representation Layer. `CustomerAddress` could compile without seeing any identifiers from our GUI Toolkit, WTL. The GUI Layer (`ProjectDlg`) could compile without seeing any of the XML library identifiers. Actually moving the classes into separate files should be a trivial formality. As the Representation Layer grows, a real project should give it independent tests. Each source file would `#include` fewer headers.

We call it a “Representation” Layer because if the XML data changes its schema, then the Representation Layer should change its innards, but not the schema it presents to clients. It should insulate `ProjectDlg`, preventing changes from rippling into it.

Regulate the MS Windows Event Queue

For our next feature, we must write on an edit field, then force `ProjectDlg` and `CustomerAddress` to save the changed text. That requires a Temporary Visual Inspection to see if our code to Set an edit field works the same as manually writing on it. We will write half a test. It will change the text, then stop reveal our dialog, with the new text in it.

An MS Windows window displays by responding to its `WM_PAINT` and related messages. To reveal the dialog, `ShowWindow()` starts the `WM_PAINT` sequence; `::GetMessage()` pulls messages from the queue, and `::PostMessage()` sends them to their target windows. MS Windows provides two functions, instead of one `mainloop()`, so programmers can intercept messages for special treatment, before their windows receive them. *Temporary Visual Inspection* and *Manual Test* fixtures for windows that require extra message preprocessing will be more complex than the simplest one we develop here.

Calling the fixture:

```
TEST_(TestDialog, address_1)
{
    CPPUNIT_ASSERT_EQUAL( "Heppy Land",
        m_aDlg.getText(IDC_EDIT_ADDRESS_1) );

    reveal();
}
```

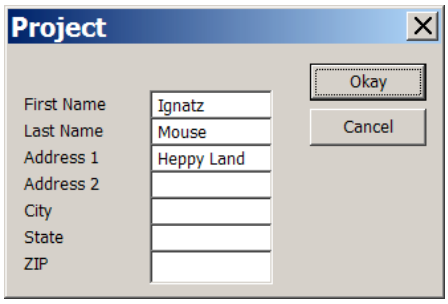
Implementing the fixture:

```
    void
TestDialog::reveal()
{
    m_aDlg.ShowWindow(SW_SHOW);
    ::SetForegroundWindow(m_aDlg);
    MSG msg;

    for(;;)
        {
        BOOL bRet = ::GetMessage(&msg, NULL, 0, 0);

        if (!bRet && msg.hwnd == m_aDlg)
            break;

        ::DispatchMessage(&msg);
        }
}
```

Between `GetMessage()` and `DispatchMessage()`, `reveal()` responds to a `WM_QUIT` message by turning the loop off. An ordinary program would exit, but test cases can continue to manipulate target window after the user closes it. `DispatchMessage()` passes surviving `MSG` records into the message's target window and its event handler.

Our `reveal()` neglects the message preprocessing system. To decouple the various ways your windows need to pre-process messages from your unique application message loop, MS Windows provides `PreTranslateMessage()`. A more complete `reveal()` system here would use WTL's `CMessageLoop` wrapper on `PreTranslateMessage()`, but our simple controls won't need it.

Without the call to `SetForegroundWindow()`, the window manager would assume the user (you) wanted to see whatever window was in front, and the new target window might appear behind your editor. Pushing the test window to the foreground obeys the *One Button Testing* Principle. No further manipulations, after launching the tests, are needed to see the target window:



And clicking on `Okay`, or `Cancel`, or the little `X` in the upper right corner won't make it go away! This *Temporary Visual Inspection* is now a *Temporary Interactive Test*, revealing our event handlers must close the window. While they don't, they are both user- and programmer-hostile. The rules for *Event Queue Regulation* claim closing a window should resume testing.

To fix that bug, we only need to research how MS Windows commonly works. The solution won't use any hard logic, so we can just author the feature, manually check it works, disable the *Temporary Interactive Test*, and keep going:

```
BEGIN_MSG_MAP(ProjectDlg)
    MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
    COMMAND_ID_HANDLER(IDOK, OnCancel)
    COMMAND_ID_HANDLER(IDCANCEL, OnCancel)
END_MSG_MAP()

LRESULT OnCancel(UINT uCode, int nID, HWND, int)
{
    if (m_bModal)
        EndDialog(nID);
    else
        PostMessage(WM_QUIT);

    return 0;
}
```

Both branches of that `if` statement supply the `WM_QUIT` that exits `reveal()`. But why is the `if` statement there?

Don't Mode Me In

MS Windows supports two kinds of dialog windows—modal and modeless. WTL launches the former with `DoModal()`; this blocks the user from clicking a dialog's owner window. And such dialogs block control flow inside the program. `DoModal()` won't return until the dialog closes, so the remaining statements in your function will block until then. This permits a dialog to request the user to select a command button—`OK`, `Cancel`, etc.—and the calling function can then immediately use the returned value. So only `EndDialog()` may dismiss a dialog invoked by `DoModal()`—to supply its return value.

We didn't write `DoModal()` itself, so it needs less testing. If it breaks, we can blame our vendor. But we can *Temporarily Interactively Test* `DoModal()` like this:

```
TEST_(TestCase, DoModal)
{
    ProjectDlg aDlg(xml);
 //  aDlg.DoModal();
}
```

If you de-comment the second line, put a breakpoint on the first line, and step the debugger through that function, you must close the dialog before control flow can reach the final brace `}`. Don't integrate tests that raise a blocking window. It would block an unattended test run.

MS's design couples modality to event blocking, permitting programmers to write less than strictly Event-Driven code as a convenience. Modal dialogs are harder to test. To write such a case, we would have to give `ProjectDlg` a pointer to a function. Its `OnInitDialog()` handler would call that function to call test code and close the window. The production code would assign an empty function to that pointer. If a test suite intends to constrain that dialog's controls behaviors, getting the dialog's exact launch sequence correct, during test, may or may not make those tests more accurate.

Our simpler tests `Create()` the dialog in its modeless configuration. So our tests don't strictly use the dialog the same way the calling code might. To assist the tests, `OnCancel()` detects which system created the dialog—either our `reveal()`'s `ShowWindow()`, or WTL's `DoModal()`. Then `OnCancel()` picks the correct exit method accordingly.

TFUI test fixtures are extra code to add testability to GUI Toolkits, but sometimes that extra code appears inside production code. If it were larger than a couple of lines, or if it linked to test code functions, we must conditionally compile it with the `_DEBUG` macro. Integration tests should compile all different versions of a program, including both Debug Mode and Release Mode, to ensure refactors in one mode don't break other modes.

Those extra lines inside the tested dialog finish our Temporary Visual Inspection feature. One can call `reveal()` at any point in any test case, even cases more complex than ours. After inspecting and dismissing the dialog, any remaining test lines will continue to function. But these windows might pop up and annoy our colleagues working on their features.

Continuous Integration

These test fixtures lead to noisy tests. Windows pop up, waving for attention, and tests don't resume until I close them.

I want to commit code changes to a team's common codebase without interrupting my own Flow. My team needs my commits to not interrupt their flow. Before committing, I don't want to search for `reveal()` instances. I want `reveal()` to only show the tested dialog to me; not my colleagues. I want `reveal()` to search for me:

```
TEST_(TestDialog, address_1)
{
    CPPUNIT_ASSERT_EQUAL( "Heppy Land",
        m_aDlg.getText(IDC_EDIT_ADDRESS_1) );

    revealFor("phlip");
}
```

Modesty forbids me calling `reveal("phlip")`, so I added "`For`". The implementation relies on the process environment:

```
    void
TestDialog::revealFor(CString user)
```

```
    {
        if (getenv("USERNAME") != user)
            return;

        m_aDlg.ShowWindow(SW_SHOW);
…
    }
```

Now I can safely commit without interrupting my colleagues' test runs. Unless if they spoof me.

Name that Pattern

In a moment, we will write a test case that inputs text into an edit field, then forces `ProjectDlg` and `CustomerAddress` to save the changed value. But those objects shouldn't save incomplete data, so first we'll finish our `ProjectDlg`'s data output features. This is now trivial. Add more fields to the XML, and add tests that query them all out:

```
            char const *
        xml = "<user>"
                "<first_name>Ignatz</first_name>"
                "<last_name>Mouse</last_name>"
                "<address_1>Heppy Land</address_1>"
                "<address_2></address_2>"
                "<city>Coconino County</city>"
                "<state>AZ</state>"
                "<zip>86351</zip>"
                "</user>";

    TEST_(TestDialog, address_2)
    {
        CPPUNIT_ASSERT_EQUAL( "",
            m_aDlg.getText(IDC_EDIT_ADDRESS_2) );
    }

    TEST_(TestDialog, city)
    {
        CPPUNIT_ASSERT_EQUAL( "Coconino County",
            m_aDlg.getText(IDC_EDIT_CITY) );
    }

    TEST_(TestDialog, state)
    {
        CPPUNIT_ASSERT_EQUAL( "AZ",
            m_aDlg.getText(IDC_EDIT_STATE) );
    }

    TEST_(TestDialog, zip)
    {
        CPPUNIT_ASSERT_EQUAL( "86351",
            m_aDlg.getText(IDC_EDIT_ZIP) );
    }
```

Before passing the tests, some more notes on style and robustness.

Ubiquitous Language Works at All Scales

The names of the tests, the names of the XML fields, the ends of the resource identifiers, and the prompt labels all use *exactly* the same words. `zip` is `<zip>` is `_ZIP` is "`ZIP`". We did not call one of them `PostalCode`, or `ZC`, or anything that would break continuity with the others. The book *Domain Driven Development*, by Eric Evans, calls this practice "Ubiquitous Language" when it extends into areas that traditional Extreme Programming calls the "System Metaphor".

If the Customer Team requests different prompt labels (within the engineers' locale), the other identifiers should change too, maximizing self-documentation, all the way down through the Logic Layer. A global search for "zip" should reveal the variable's path through the entire system.

Changing "zip" is a good idea, because an industrial-strength application should not assume all Customer Addresses are inside the USA, or that they use USA-style Zip Codes. Our design will soon be flexible enough that such territory extensions become easy.

Boundary Conditions

Also, note the `<address_2>` node is empty. While making the tests pass, `CustomerAddress::get()` crashed, and quickly revealed it needed an upgrade. Now it checks for a `NULL` pointer to an `XMLNodePtr_t` before using it:

```
    _bstr_t
CustomerAddress::get(_bstr_t field)
{
    XMLNodePtr_t pField =
     m_pXML->selectSingleNode("/user/" + field + "/text()");

    return pField.GetInterfacePtr()? pField->text: "";
}
```

That function uses the narrowest XPath possible, with `"/text()"` on the end, to select the required attribute of the target node. The XML standard declares that parsers cannot distinguish `<address_2></address_2>` from `<address_2/>`, so either would bear no object for `text()` to return. Nodes with no contents inspire `selectSingleNode()` to return a `pField` with no COM object assigned to it. This requires us to check for `NULL`.

And here's the code to pass the tests:

```
    LRESULT
ProjectDlg::OnInitDialog(UINT, WPARAM, LPARAM, BOOL &)
{
    CustomerAddress aCA(m_xml);
    SetDlgItemText( IDC_EDIT_FIRST_NAME, aCA.get("first_name") );
    SetDlgItemText( IDC_EDIT_LAST_NAME,  aCA.get("last_name")  );
    SetDlgItemText( IDC_EDIT_ADDRESS_1,  aCA.get("address_1")  );
    SetDlgItemText( IDC_EDIT_ADDRESS_2,  aCA.get("address_2")  );
    SetDlgItemText( IDC_EDIT_CITY,       aCA.get("city")       );
    SetDlgItemText( IDC_EDIT_STATE,      aCA.get("state")      );
    SetDlgItemText( IDC_EDIT_ZIP,        aCA.get("zip")        );
    return 0;
}
```

That looks more and more like a latent table, waiting to come out.

To force the issue, our next test writes on an edit field. The we'll upgrade `ProjectDlg` and `CustomerAddress` to save the changed text. The data structure will become interactive. Until

now it only displayed XML. If we add a test that requires the dialog to save data changes back into the XML, then we might need to duplicate that table.

I wrote the test we need, but then I commented-out everything that doesn't work, except for one line. A *Temporary Visual Inspection* checks this test's intermediate state, before finishing it. We will get that (simple) line to work, and un-comment the other lines:

```
TEST_(TestDialog, changeName)
{
    m_aDlg.SetDlgItemText(IDC_EDIT_FIRST_NAME, "Krazy");
    m_aDlg.SetDlgItemText(IDC_EDIT_LAST_NAME,  "Kat");

    revealFor("phlip");

    CustomerAddress &aCA = m_aDlg.getCustomerAddress();
  //  m_aDlg.saveXML();
//CPPUNIT_ASSERT_EQUAL(_bstr_t("Krazy"), aCA.get("first_name") );
//CPPUNIT_ASSERT_EQUAL(_bstr_t("Kat"),   aCA.get("last_name")  );
}
```

I marked features that don't exist yet in **`bold`**. The test begins by simulating a user changing the fields. Edit fields with complex behaviors, such as customized keystrokes or elaborate formatting, should test using `SendMessage(WM_CHAR, … )` and similar low-level input events. This project only needs the simple behaviors. We did not invent the edit field itself.

To supply `getCustomerAddress()`, we refactor `aCA` out of `OnInitDialog()`, and replace `m_xml` with it:

```
    class
ProjectDlg:
    public CDialogImpl<ProjectDlg>
{
…
    ProjectDlg(char const *xml): m_aCA(xml) {}
    CustomerAddress &getCustomerAddress() { return m_aCA; }

  private:
    CustomerAddress m_aCA;
};
```

But to make the rest of the test to pass, we must write `saveXML()`, and this must call a new method on `m_aCA` to `set()` its XML data contents.

Because MSXML is still strange to us, we backed off and commented-out much of the recent test. Now write a new Child Test, to teach `CustomerAddress` how to `set()`:

```
TEST_(TestCase, set)
{
    CustomerAddress aCA(xml);
    aCA.set("first_name", "Offisa");
    aCA.set("last_name",  "Pup");
 CPPUNIT_ASSERT_EQUAL(_bstr_t("Offisa"), aCA.get("first_name") );
 CPPUNIT_ASSERT_EQUAL(_bstr_t("Pup"),    aCA.get("last_name")  );
}
```

That's our first direct test on the new `CustomerAddress` class. (Note the `TEST_()` macro supplies `TestCase`, not `TestDialog`.) A real software project would have already added tests for its existing behaviors, out of curiosity, and to prove it's test-worthy.

A minor bug in `_bstr_t` prevents `operator==` from working cleanly on string literals, so the `ASSERT_` statements must construct `_bstr_t`s for the comparisons. That explains the `_bstr_t("Offisa")` inside `CPPUNIT_ASSERT_EQUAL()`.

To pass the test, a new method set() must locate the target node, and assign a new value to it. MSXML provides miniature database facilities:

```
    class
CustomerAddress
{
  public:
    CustomerAddress(CString xml);
    _bstr_t get(_bstr_t field);
    void    set(_bstr_t field, TCHAR const * nuText);

  private:
    IXMLDOMDocument2Ptr m_pXML;
};

    void
CustomerAddress::set(_bstr_t field, TCHAR const * nuText)
{
    XMLNodePtr_t pField =
        m_pXML->selectSingleNode("/user/" + field);

    pField->text = nuText;
}
```

Now we enable the suspended lines in `TEST_(TestDialog, changeName)`, and write `saveXML()`:

```
    void
ProjectDlg::saveXML()
{
    m_aCA.set("first_name" , getText( IDC_EDIT_FIRST_NAME ));
    m_aCA.set("last_name"  , getText( IDC_EDIT_LAST_NAME  ));
    m_aCA.set("address_1"  , getText( IDC_EDIT_ADDRESS_1  ));
    m_aCA.set("address_2"  , getText( IDC_EDIT_ADDRESS_2  ));
    m_aCA.set("city"       , getText( IDC_EDIT_CITY       ));
    m_aCA.set("state"      , getText( IDC_EDIT_STATE      ));
    m_aCA.set("zip"        , getText( IDC_EDIT_ZIP        ));
}
```

After that passes its test, we take a crack at duplication again. Both `saveXML()` and `OnInitDialog()` show the same list of inputs to different functions. So we pull the inputs out and make a single explicit table out of the two implicit ones:

```
    struct
field {  CString m_name; UINT m_id;  } fields[] = {
        { "first_name" , IDC_EDIT_FIRST_NAME },
        { "last_name"  , IDC_EDIT_LAST_NAME  },
        { "address_1"  , IDC_EDIT_ADDRESS_1  },
        { "address_2"  , IDC_EDIT_ADDRESS_2  },
        { "city"       , IDC_EDIT_CITY       },
        { "state"      , IDC_EDIT_STATE      },
        { "zip"        , IDC_EDIT_ZIP        },
    };

    void
ProjectDlg::saveXML()
```

```
{
    for (int x (0);  x < sizeof fields / sizeof fields[0];  ++x)
        m_aCA.set
            (
            fields[x].m_name.GetBuffer(0),
            getText(fields[x].m_id)
            );
}

    LRESULT
ProjectDlg::OnInitDialog(UINT, WPARAM, LPARAM, BOOL &)
{
    for (int x (0);  x < sizeof fields / sizeof fields[0];  ++x)
        SetDlgItemText
            (
            fields[x].m_id,
            m_aCA.get(fields[x].m_name.GetBuffer(0))
            );
    return 0;
}
```

Some code got uglier, and some got cleaner. One benefit is we can add new fields in one place, not two. That will be useful when the Post Office invents a new kind of address.

The next refactor trades two `fields[x]` expressions for one:

```
    struct
Field
{
    CString m_name;
    UINT    m_id;

void dialogFromXML(ProjectDlg &aDlg, CustomerAddress &aCA)
    {
    aDlg.SetDlgItemText(m_id, aCA.get(m_name.GetBuffer(0)));
    }

void dialogToXml(ProjectDlg &aDlg, CustomerAddress &aCA)
    {
    aCA.set(m_name.GetBuffer(0), aDlg.getText(m_id));
    }
};

Field fields[] = {
        { "first_name" , IDC_EDIT_FIRST_NAME },
...
        { "zip"        , IDC_EDIT_ZIP        },
    };

    void
ProjectDlg::saveXML()
{
    for (int x (0);  x < sizeof fields / sizeof fields[0];  ++x)
        fields[x].dialogToXML(*this, m_aCA);
}

    LRESULT
ProjectDlg::OnInitDialog(UINT, WPARAM, LPARAM, BOOL &)
{
    for (int x (0);  x < sizeof fields / sizeof fields[0];  ++x)
        fields[x].dialogFromXML(*this, m_aCA);
```

```
    return 0;
}
```

That `Field` `structure` looks like it's just about to turn into a `class`…
To summarize our current architecture, we now have…

- A dialog to view the data
- An XML repository to model the data
- A `Field` `structure` declaring a link between the view and model.

Eventually, that `Field` structure will upgrade from a `structure` to a `class`, and behavior will migrate inside it. It will grow from declaring the link to controlling the link between the view and model.

I will next remove duplication in a way that pushes the boundaries of my C++ Sane Subset. On a team, only write code like this if the team follows Pair Programming, and if your pair likes the code. This technique is another example of the tension between the "*Remove Duplication* Simplicity Principle and the *Clear and Expressive Code* Principle. In some situations, the technique provides a higher-level abstraction. In other situations, it obfuscates mercilessly.

Member Function Pointers

I put the arguments to the methods `xmlToDialog()` and `dialogFromXML()` in the same order for a reason. (If I had forgotten to put the arguments in the same order, I'd refactor in one more step here.)

We want only one function to loop through the fields. That's a noble goal, but C++ only lets us do it by replacing direct calls to `xmlToDialog()` and `dialogFromXML()` with a call through a member function pointer. That requires both methods to use the same signature, with the parameters in the same order. That's why I used "`To`" in one function but "`From`" in the other.

Here's the code refactored so the `for` loop only appears in one place:

```
    void
ProjectDlg::transferXML
    (
    void (Field::*transfer)( ProjectDlg      &aDlg,
                             CustomerAddress &aCA )
    )
{
    for (int x (0);  x < sizeof fields / sizeof fields[0];  ++x)
        (fields[x].*transfer)(*this, m_aCA);
}

    void
ProjectDlg::saveXML()
{
    transferXML(&Field::dialogToXML);
}

    LRESULT
ProjectDlg::OnInitDialog(UINT, WPARAM, LPARAM, BOOL &)
{
    transferXML(&Field::dialogFromXML);
    return 0;
}
```

That technique is the Execute Around Pattern in C++. Languages with block closures make the pattern look cleaner. In C++, we could dabble in `std::for_each()`, providing about the same ratio of abstraction to clarity.

However, our implementation of `OnInitDialog()` is now much shorter than the average for that method!

This Case Study's future refactors and features will not strictly require this technique. Only use member function pointers on a real project if your team agrees they need it. This book needs it as a cautionary tale against suppressing *all* duplication.

Organic designers merge duplication under the mild assumption that new features will reuse the resulting abstraction. In this case, `transferXML()`'s abstraction is "Input / Output". That's as old as the hills, and computer scientists are not likely to discover a third direction any time soon. But when they do, our design will be ready.

To see if all this refactoring made the design better, not worse, add a new requirement. (Honestly, all we can hope to show is that these refactorings have made the design easy to refactor *again* to make it better.)

Let's say our Customer Team wants the State code to upgrade from an edit field to a drop-down combo box. The XML will offer a list of states as part of the `<state>` node:

```
<state options='AK,AL,AR,AZ,'>AZ</state>
```

To save room, our clever marketing department only targets USA States beginning with A. We will let them see the consequences of their decision, very soon.

But our current design squeezes six edit fields into two functions, each of which assumes all the controls are edit fields! How can we change just one of them into a combo box?

Deprecation Refeaturization

Our dialog displays a list of edit fields. One of them must become a combo box. The source code supporting those edit fields has aggressively refactored together into a table of `Field` structures. Adding another edit field would be easy, but adding a combo box would be impossible. Each `Field` structure assumes its target is an edit field.

Eventually, that table will be polymorphic, and will control edit fields and combo boxes behind the same interface. But we should not introduce this new design in one big jump. It would require more than 10 edits between passing tests. Passing tests are more important than a clean design. If we start with the new behavior, then refactor under test toward that improved design, the tests will ensure the design supports our behavior.

The first step of our strategy is to add a new combo box, next to the edit field, and leave them both online for a while. The combo box, and its support code, will not participate in the Field table until its behavior is finished.

This strategy leaves the old feature—the "state" edit field—online while creating a new one. This sloppy Deprecation Refactor permits behavior to slide into new features. The first step adds a new, redundant feature without any regard to the existing code.

Write a test that forces a new combo box to exist:

```
    #include <atlCtrls.h>
…
```

```
TEST_(TestDialog, IDC_COMBO_STATE)
{
    CComboBox aBox(m_aDlg.GetDlgItem(IDC_COMBO_STATE));
    COMBOBOXINFO cbi = { sizeof cbi };
    BOOL isComboBox = aBox.GetComboBoxInfo(&cbi);
    CPPUNIT_ASSERT(isComboBox);
}
```

For most projects, you won't so aggressively test that you really did author a combo box. This test could proceed to verify every member of the structure `COMBOBOXINFO`. In practice, other tests provide cross-coverage, reducing the odds our combo box mutates into something else unnoticed.

A language note: I did not write `CPPUNIT_ASSERT_EQUAL(TRUE, isComboBox)` because most boolean systems in the C languages use any non-zero value for `TRUE`, so the statement `TRUE == isComboBox` could return a false negative. `CPPUNIT_ASSERT_EQUAL()` would expand to that statement, so I quickly added `CPPUNIT_ASSERT()`:

```
#define CPPUNIT_ASSERT(boolean)                          \
    if (!(boolean))  {  stringstream out;                \
        out << __FILE__ << "(" << __LINE__ << ") : "; \
        out << #boolean << "(" << (boolean) << ") != "; \
        cout << out.str() << endl;                       \
        OutputDebugStringA(out.str().c_str());           \
        OutputDebugStringA("\n");                        \
        all_tests_passed = false;                        \
        __asm { int 3 }  }
```

To pass the new test, add the combo box to the raw resource file. We temporarily put it to the right of the edit field that it intends to replace:

```
LTEXT         "State",IDC_STATIC,7,68,45,10
EDITTEXT      IDC_EDIT_STATE,61,67,53,13,ES_AUTOHSCROLL
COMBOBOX      IDC_COMBO_STATE,128,66,51,50,CBS_DROPDOWNLIST |
                  CBS_SORT | WS_VSCROLL | WS_TABSTOP
LTEXT         "ZIP",IDC_STATIC,7,78,45,10
EDITTEXT      IDC_EDIT_ZIP,61,77,53,13,ES_AUTOHSCROLL
```

(Inside MS Windows, combo boxes conjoin list boxes and edit fields, so at this juncture we could have simply changed `IDC_EDIT_STATE`'s type into a combo box. All the WTL statements which think that `IDC_EDIT_STATE` is an edit field would continue to work properly. Our Deprecation Refeaturization could have been much easier. To reveal why the previous pages claimed to improve design despite forcing all controls to be edit fields, this Case Study will pretend that our Customer Team has specified the user cannot enter a state code by typing letters into the field; only by scrolling the list. That requires the combo box to use the `CBS_DROPDOWNLIST` bit, which forces a bigger change.)

Notice we left the old `IDC_EDIT_STATE` edit field in place. It is now deprecated—scheduled for retirement. Some deprecations take longer than the integration cycle, or even the release cycle. If integration interrupts a refeaturization, write comments on statements that reference the deprecated identifier (here `IDC_EDIT_STATE`), warning our colleagues not to use it. If deprecating a GUI features takes longer than the delivery cycle, make the deprecated controls invisible.

Split

To put a list of states into the combo box, we need a function that gets the comma-delimited list of states out of the XML, and one that puts a list of states into a combo box. The state list will transport as a comma-delimited string.

Our first function must convert a comma-delimited string into a "Standard Template Library" container, `std::list< std::CString >`. Some environments provide a `split()` function out-of-the-box. If your C++ project already has a `split()`, this part would be easier:

```
    typedef std::list< CString > CStrings_t;

    TEST_(TestCase, split)
    {
        CStrings_t list = split("tiger tea thunder", ' ');
        CPPUNIT_ASSERT_EQUAL(3, list.size());
        CStrings_t::const_iterator it = list.begin();
        CPPUNIT_ASSERT_EQUAL("tiger", *it);  ++it;
        CPPUNIT_ASSERT_EQUAL("tea",    *it);  ++it;
        CPPUNIT_ASSERT_EQUAL("thunder", *it);
    }
…
        CStrings_t
    split(CString str, char delim)
    {
        CStrings_t list;

        for (;;)
            {
            int at = str.Find(delim);
            if (at == -1)  break;
            list.push_back(str.Mid(0, at));
            str = str.Mid(at + 1);
            }
        if (str.GetLength())
            list.push_back(str);

        return list;
    }
```

Next, retrieve the list of states from the XML:

```
    TEST_(TestCase, getStateList)
    {
        CustomerAddress aCA(xml);
        CString stateList = aCA.getStateList();
        CPPUNIT_ASSERT_EQUAL("AK,AL,AR,AZ,", stateList);
    }


        CString
    CustomerAddress::getStateList()
    {
        XMLNodePtr_t pField =
            m_pXML->selectSingleNode("/user/state/@options");

        return pField->text.operator LPCSTR();
    }
```

That's enough support; the next test forces the combo box to populate. After `TestDialog::setUp()` creates the dialog, indirectly calling `ProjectDlg::OnInitDialog()`, this

case calls `GetLBText()` to fetch the text out of each list box item. Then assertions check they are the correct state codes:

```
TEST_(TestDialog, populateComboBox)
{
    CComboBox aBox(m_aDlg.GetDlgItem(IDC_COMBO_STATE));
    CString str;
    aBox.GetLBText(0, str);  CPPUNIT_ASSERT_EQUAL("AK", str);
    aBox.GetLBText(1, str);  CPPUNIT_ASSERT_EQUAL("AL", str);
    aBox.GetLBText(2, str);  CPPUNIT_ASSERT_EQUAL("AR", str);
    aBox.GetLBText(3, str);  CPPUNIT_ASSERT_EQUAL("AZ", str);
    CPPUNIT_ASSERT_EQUAL(4, aBox.GetCount());
}
```

To pass the test, we simply cram new behavior into `OnInitDialog()`, turning it from clean to crufty again:

```
    LRESULT
ProjectDlg::OnInitDialog(UINT, WPARAM, LPARAM, BOOL &)
{
    transferXML(&Field::dialogFromXML);

    CString stateList = m_aCA.getStateList();
    CStrings_t states = split(stateList, ',');
    CComboBox aBox(GetDlgItem(IDC_COMBO_STATE));

    for ( CStrings_t::iterator it(states.begin());
          it != states.end();  ++it )
        {
        int at = aBox.AddString(*it);
        assert(at > -1);
        }
    return 0;
}
```

That new code in `OnInitDialog()` now duplicates the concept of the `Field` controllers. The older code populates controls, and the new code populates a control in a different way. The code doesn't look the same, but it ought to hide behind the same interface. The design is, again, half one style and half another.

Polymorphic Smart Pointer Array

Upgrade the table of controllers, to get them ready for polymorphism. First, convert the `Field` from a `structure` to a `class`, with methods ready to override. C++ `structures` have all the same abilities `classes` do, but we pretend they don't. Our C++ Sane Subset uses `struct` only to mean "data bucket", and `class` to mean "behavior bucket". Our intermediate design stretched that guideline.

```
    class
Field
{
  public:
    Field(char const * name, UINT id):
        m_name(name),
        m_id(id)
            {}

    virtual void dialogFromXML( ProjectDlg      &aDlg,
```

```
                                CustomerAddress &aCA );

    virtual void dialogToXML( ProjectDlg      &aDlg,
                              CustomerAddress &aCA );

  protected:
    CString const m_name;
    UINT    const m_id;
};

Field fields[] = {
        Field( "first_name" , IDC_EDIT_FIRST_NAME ),
        Field( "last_name"  , IDC_EDIT_LAST_NAME  ),
...
        Field( "zip"        , IDC_EDIT_ZIP        ),
    };
```

Part of learning a Sane Subset, especially for C++, is learning what statements bring which hidden side effects. That array of `fields` quietly calls a default copy constructor, which C++ defines as a copy of each member. C++ is very mechanical, and cannot simply fling objects about the way softer languages can. Only the keyword `new` creates a C++ heap object.

C++ Sane Subsets should specify explicit copy constructors, to prevent problems such as copies of raw pointers leading to memory corruption. We know our little class only uses members that themselves have valid copy constructors, so Sane Subset compliance stretches, again. If this code weren't temporary, we would provide a real, explicit copy constructor. But because we are about to derive a type from that class, we must change the instantiation system to accommodate polymorphism. Copy constructors, even explicit ones, cannot reliably copy objects of derived classes, so the code soon won't use them.

Notice `m_name` and `m_id` are now `protected`. Formerly, when they lived in a `structure`, they were `public`. Previous refactors serendipitously ensured only methods of `Field` accessed those member data. For a few pages now they have been *de-facto* `private`.

Copy-constructing an array of polymorphic objects would slice our potential `ComboField` controllers into `Field` controllers. The array must not store objects but pointers, and something must `delete` the target objects when the program ends:

```
#include <memory>

typedef std::auto_ptr<Field> FieldPtr_t;

    FieldPtr_t static
fields[] = {
    FieldPtr_t(new Field( "first_name" , IDC_EDIT_FIRST_NAME )),
    FieldPtr_t(new Field( "last_name"  , IDC_EDIT_LAST_NAME  )),
    FieldPtr_t(new Field( "address_1"  , IDC_EDIT_ADDRESS_1  )),
    FieldPtr_t(new Field( "address_2"  , IDC_EDIT_ADDRESS_2  )),
    FieldPtr_t(new Field( "city"       , IDC_EDIT_CITY       )),
    FieldPtr_t(new Field( "state"      , IDC_EDIT_STATE      )),
    FieldPtr_t(new Field( "zip"        , IDC_EDIT_ZIP        )),
    };
```

The Flyweight Pattern, in C++, requires those objects to exist somewhere in storage. And `auto_ptr<>` provides a healthy copy constructor. It transfers ownership into our file-`static` array, and allows `fields` to stop calling `Field`'s default copy constructor.

Our quest to fold duplication created a plethora of `auto_ptrs`. The mantra "remove behavioral duplication" can appear specious. Some duplicated behaviors are more equal than

others. We have indeed minimized the behaviors that we must edit and maintain, at the expense of much structure—all the `[] {}` and `()` delimiters—and of much behavior inside the C++ Standard Library.

`auto_ptr`'s constructor forces us to `explicit`ly call `FieldPtr_t()` on each array element. That adds to the appearance of duplication, but it's just `auto_ptr`'s way of reminding us that its argument must be a `new` heap object. The *Exceptional C++* books by Herb Sutter cover these topics in great detail; we may eventually find a way around all that clutter.

We could have written a custom smart pointer. We could have declared a `static` instance of each Field type, and then stored an array of their addresses. No `new` or `delete`. We could have wrapped `fields[]` into a `static` object with its own destructor, to free the `*Field`s when the process ends. We could have just leaked all the `Field`s, and let the MS Windows Memory Fairy magically free them. I picked the above solution to imitate the style of polymorphic arrays in languages with garbage collection.

Attempting to compile, we discover `auto_ptr<>` does not provide `operator->*()`. That is probably a good thing. Add to `transferXML()` an assignment into a real pointer, and use that to call the target operation:

```
    void
ProjectDlg::transferXML
    (
  void (Field::*transfer)(ProjectDlg &aDlg, CustomerAddress &aCA)
    )
{
 for (int x (0);  x < sizeof fields / sizeof fields[0];  ++x)
        {
        Field * pField = fields[x].get();
        (pField->*transfer)(*this, m_aCA);
        }
}
```

We have "improved" the design (for a fairly wide definition of "improve"), while running the tests over and over again between each tweak. Now we throw in a new class to handle combo boxes:

```
    class
ComboField: public Field
{
  public:
    ComboField(char const * name, UINT id):
        Field(name, id)
            {}
};

    FieldPtr_t static
fields[] = {
    FieldPtr_t(new Field( "first_name" , IDC_EDIT_FIRST_NAME )),
    FieldPtr_t(new Field( "last_name"  , IDC_EDIT_LAST_NAME  )),
    FieldPtr_t(new Field( "address_1"  , IDC_EDIT_ADDRESS_1  )),
    FieldPtr_t(new Field( "address_2"  , IDC_EDIT_ADDRESS_2  )),
    FieldPtr_t(new Field( "city"       , IDC_EDIT_CITY       )),
    FieldPtr_t(new ComboField( "state" , IDC_EDIT_STATE      )),
    FieldPtr_t(new Field( "zip"        , IDC_EDIT_ZIP        )),
    };
```

It does not do much, yet. But the tests still pass.

Let's try overriding a method. In tiny steps, its implementation simply calls the parent's method. We are building a location to copy some statements from `OnInitDialog()` to:

```
    class
ComboField: public Field
{
  public:
    ComboField(char const * name, UINT id):
        Field(name, id)
            {}

    void dialogFromXML( ProjectDlg      &aDlg,
                        CustomerAddress &aCA );
};

    void
ComboField::dialogFromXML( ProjectDlg      &aDlg,
                           CustomerAddress &aCA )
{
    Field::dialogFromXML(aDlg, aCA);
}
```

Again, the tests still pass.

Notice that when `Field` first appeared we didn't make it into a true class, with behavior, until finding a real reason. Only that reason can assist giving `Field` the right features. We did not speculate about any features it might need, or throw all possible `Field`-like concepts in.

Similarly, when we finish our new `ComboField` class, it won't re-usably channel data from different sources. It will only hard-code the call to `aCA.getStateList()`. When we find a reason to upgrade that, the reason itself will show us the way to supply the combo box's list.

Those changes forced the `Fields` list to metamorphosize from a homogeneous into a heterogeneous array. Future `Fields` lists can plug and play other kinds of controls more easily.

Move the extra code out of `OnInitDialog()` and pop it in here (and also select the current state):

```
    void
ComboField::dialogFromXML( ProjectDlg      &aDlg,
                           CustomerAddress &aCA )
{
    Field::dialogFromXML(aDlg, aCA);
    CString stateList = aCA.getStateList();
    CStrings_t states = split(stateList, ',');
    CComboBox aBox(aDlg.GetDlgItem(IDC_COMBO_STATE));

    for ( CStrings_t::iterator it(states.begin());
          it != states.end();  ++it )
        {
        int at = aBox.AddString(*it);
        assert(at > -1);
        }
    CString selectedState = aCA.get(m_name);
    aBox.SelectString(-1, selectedState);
}
```

The tests still pass.

(Along the way I hid `_bstr_t` inside `CustomerAddress`. It now only shows `CString` at its interfaces, so we can take out some odious `.GetBuffer(0)` calls and similar conversion

functions. Projects should generally pick only one of the many string classes available, and stick with it in all public interfaces. And `_bstr_t` has too few features.)

That new method must not hard-code `IDC_COMBO_STATE`. The data member `m_id` should transmit `IDC_COMBO_STATE` into it:

```
   FieldPtr_t(new ComboField( "state" , IDC_COMBO_STATE )),
…
        void
    ComboField::dialogFromXML( ProjectDlg      &aDlg,
                               CustomerAddress &aCA )
    {
    //    Field::dialogFromXML(aDlg, aCA);
        CString stateList = aCA.getStateList();
        CStrings_t states = split(stateList, ',');
        CComboBox aBox(aDlg.GetDlgItem(m_id));

        for ( CStrings_t::iterator it(states.begin());
              it != states.end();  ++it )
            {
            int at = aBox.AddString(*it);
            assert(at > -1);
            }
        CString selectedState = aCA.get(m_name);
        aBox.SelectString(-1, selectedState);
    }
```

That change broke this test (remember it?):

```
TEST_(TestDialog, state)
{
    CPPUNIT_ASSERT_EQUAL( "AZ",
        m_aDlg.getText(IDC_EDIT_STATE) );
}
```

Let's see how naïvely we can fix it:

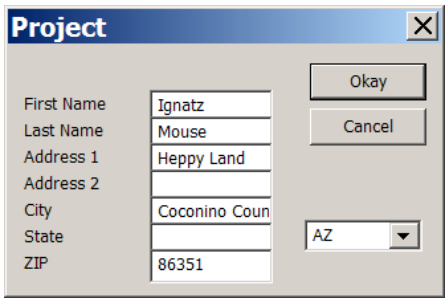
```
TEST_(TestDialog, state)
{
    CPPUNIT_ASSERT_EQUAL( "AZ",
        m_aDlg.getText(IDC_COMBO_STATE) );

    revealFor("phlip");
}
```

It works! A passing test, a *Temporary Interactive Test* of our dialog, and a code inspection reveal this single edit has finished our Deprecation Refeaturization.

Combo boxes Get their values the same way as edit fields, so `ComboField` won't need to override `dialogToXML()`. That method already calls `getText()`, due to duplication suppression, and a test case using `getText()` shows that the `getText()` inside `dialogToXML()` will work correctly too.

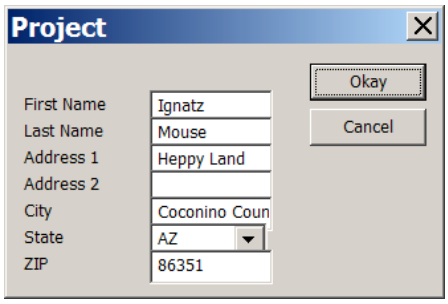
It turns out the trivial refactor to produce `getText()` (back on page 182) wasn't such a "small victory" after all…

That `revealFor()` is the first time I have looked at the dialog, since before adding the combo box. Its resource still has the original edit field, but the behavior moved over to the new combo box:



Retire the Deprecated Identifier

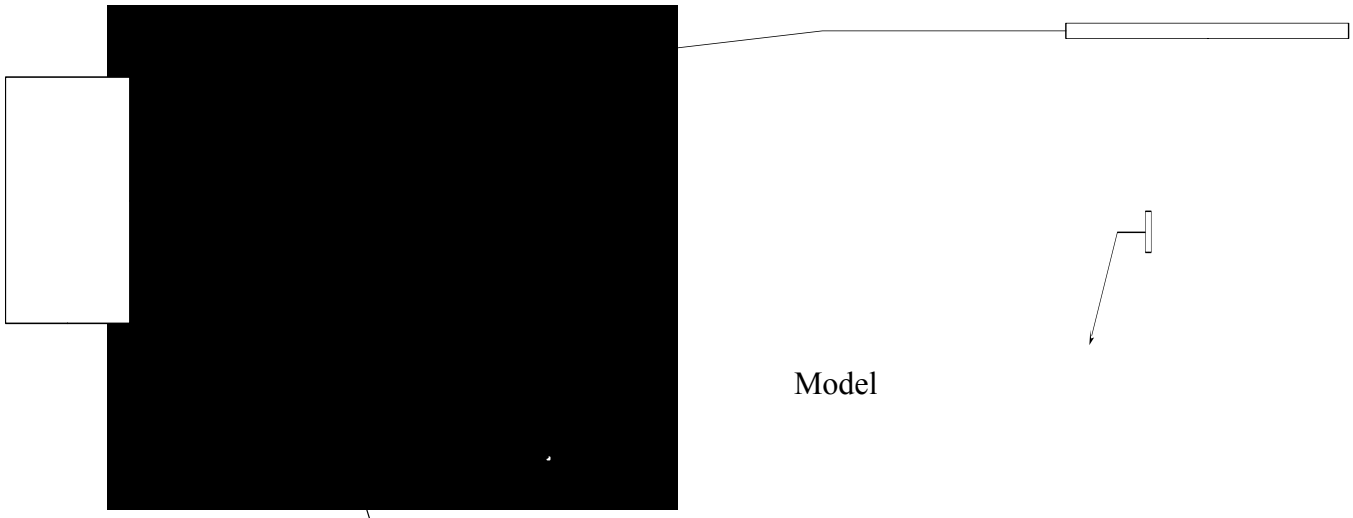
And with that last change, nobody uses `IDC_EDIT_STATE` anymore. We remove it from `resource.h` and compile. Syntax errors now force us to remove its control from `Project.rc`; and this lets us move the combo box over into its place:



When I removed `IDC_EDIT_STATE` from `resource.h`, I predicted this change would break no `.cpp` or `.h` files. Then they compiled correctly; only `resource.rc` encountered a syntax error. When you predict an error, always predict the exact error you intend, to verify your mental model of the code's state. If a broken `.cpp` or `.h` file surprised me, I would have used Undo to restore `IDC_EDIT_STATE`, then inspected those files for any remaining references to it. I would have upgraded their behaviors, incrementally, before finally retiring `IDC_EDIT_STATE`.

Dynamic Data Exchange

Our simple implementation of Model View Controller looks like this:



After such a rough chapter, its simplicity may feel anticlimactic!

If a real Logic Layer provided values for our Model, it might need to send changes in real-time to the View. A full-featured MVC links elements in each module with the Observer Pattern, forming a triangle. Ours uses direct method calls.

Our list of `Field`s form a prototypical Controller. They don't appear to control anything, but they do. Without them, `OnInitDialog()` would be full of uncontrolled cruft.

An advanced MVC implementation might require multiple objects in each module, and multiple Observer Pattern links between them. For example, the Controller module might contain a specific instance of a ComboBoxController, which observes a specific instance of a ZipCodeModel inside the Model layer. MVC is such a popular pattern because it represents a common strategy to resolve many different kinds of couplings. Your MVC triangle may differ.

Our list of `Field`s superficially resembles an MFC system, which WTL re-implements, called Dynamic Data Exchange (DDX). Wizards and programmers use it to bond controls to variables—typically to the data members of a window's wrapper class. After the user changes a control, or the application changes a variable, the wrapper's code calls `DoDataExchange()` to transfer the data one way or the other. Values copy from the member variables to the window's controls before they display, and value copy from the controls to the member variables when the user commits their change.

DDX forces window wrapper classes to *have* data member variables for each control. We didn't need them. We route data directly into their model, abstracted from the GUI Layer.

Classes hold behavior, not data.

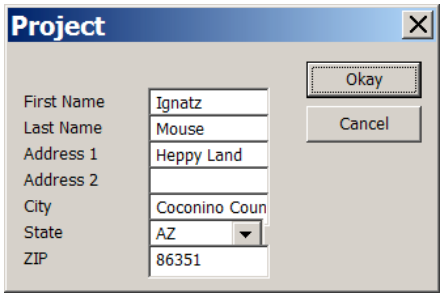
Our `Field` classes don't inherit any library classes like `CEdit`, or expose every feature of a control. Naïve programmers, following vendors' libraries as role models, might think such application-specific classes must be full-featured and "re-usable" like library classes. And they might think one inherits a class to "add methods to it". Per the *Effective C++* books by Scott Meyers, one only inherits to override a virtual method. (The exceptions are very small "convenience classes" that adjust some base class's interface without adding anything.)

Our `Field` classes are strictly application-specific. They occupy the intersection of the needs of this Representation Layer and this GUI Layer, with minimal methods that are easy to override.

This extensibility contrasts deeply with DDX, which tempts with the convenience of a table, then denies the status of full-fledged object to each “link between a GUI field and a data field”. All that wizardry leads to obese window wrapper classes stuffed with behaviors that belong in other places. That, in turn, makes re-using those classes very difficult. It is just one of the ways MFC and its tutorials encourage young programmers to use the Class Wizard to create objects instead of classes.

Chapter 9: *Broadband Feedback*

The preceding Case Study discussed internal architectures that decouple a GUI and Logic Layer. Its esthetics, however, remain thin:



All GUI Toolkits permit more colorful features, and many tests need more elaborate fixtures. This Case Study takes the project begun on page 161 in a new direction.

To move into new markets, project leaders request new technical features. To expand in established markets, they often request more esthetic glitz. This posturing showcases engineers' technical competence. For example, a charismatic leader might demand that everyone make their user-friendly software programs look just like user-hostile metal gadgets.

The previous Case Study created a project now poised to captured the "Customer Address" market. After finishing some technical details, we will add some art.

Our Customer Team requests these stories:

- Page 217: The XML persists in a file. Pass the name on the command line
- 229: A list box displays each first and last name, in columns
- 249: Localize the interface to ????????.
- 266: During long operations, a progress bar reveals the progress

All the artistic features resist testing in perverse ways. So our crack chief programmer suggests (and the Whole Team agrees with) one more iteration task:

- 271: Broadband Feedback

Fixtures will capture animations of our application testing itself, and transmit them to reviewers and domain experts.

Persistence

Beginning with the code the way the last chapter left it, we need `main()` (in non-test and Release mode) to read the XML from a file. Then we need the "Okay" button to save this file. So, after splitting my one big source file into three, and slipping in some minor upgrades, here's `Project.h`:

```
#pragma once

#include <atlStr.h>
```

```
#include <atlApp.h>

    extern CAppModule _Module;

#include <atlWin.h>
#include <atlDlgs.h>
#include <iostream>
#include <list>
#include "resource.h"
#import <msxml4.dll>

    using MSXML2::IXMLDOMDocument2Ptr;
    typedef MSXML2::IXMLDOMNodePtr XMLNodePtr_t;

    inline std::ostream &
operator<<(std::ostream &o, _bstr_t const &str)
{
    if (str.length())  o << str.operator const char *();
    return o;
}

typedef std::list<CString> CStrings_t;

    class
CustomerAddress
{
  public:
    CustomerAddress(CString xml);
    CString    get(CString field);
    void       set(CString field, TCHAR const * nuText);
    CStrings_t getStateList();
    IXMLDOMDocument2Ptr getXML()  {  return m_pXML;  }

  private:
    XMLNodePtr_t getNode(CString xPath);
    IXMLDOMDocument2Ptr m_pXML;
};

class Field;

    class
ProjectDlg:
    public CDialogImpl<ProjectDlg>
{
  public:
    enum { IDD = IDD_PROJECT };

    BEGIN_MSG_MAP(ProjectDlg)
        MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
        COMMAND_ID_HANDLER(IDOK,     OnClose)
        COMMAND_ID_HANDLER(IDCANCEL, OnClose)
    END_MSG_MAP()

    LRESULT OnClose(UINT uCode, int nID, HWND, int);
    LRESULT OnInitDialog(UINT, WPARAM, LPARAM, BOOL &);
    ProjectDlg(char const *xml): m_aCA(xml) {}
    CString getText(UINT ID) const;

    CustomerAddress &getCustomerAddress()
                        {  return m_aCA;  }
    void saveXML();

    void setFileName(CString fileName)
```

```
            {  m_fileName = fileName;  }

    void transferXML
            (
            void (Field::*transfer) ( ProjectDlg      &aDlg,
                                      CustomerAddress &aCA )
            );

  private:
    CustomerAddress m_aCA;
    CString         m_fileName;
};

typedef std::list<CString> CStrings_t;

CStrings_t split(CString str, char delim);
CString    readFile(CString fileName);

#ifdef _DEBUG
#   define ELIDE_(x) x
#else
#   define ELIDE_(x) // x
#endif

ELIDE_(bool runTests());
```

If this project had other modules, and they needed low-level things like `CStrings_t` or `ELIDE_()`, we would pull them out into their own headers.

I moved all the tests out to out to `Project_test.cpp` (not shown), and added this:

```
    bool
runTests()
{
    bool result = TestCase::runTests();

    if (result)
        cout << "All tests passed!" << endl;
    else
        cout << "Tests failed..." << endl;

    return result;
}
```

And here's `Project.cpp`:

```
#include "Project.h"  //  include this first, to test that it
                      //  includes all its required header files
#include <sstream>
#include <iomanip>
#include <atlCtrls.h>
#include <fstream>

    CAppModule _Module;

    using std::cout;
    using std::endl;
    using std::stringstream;
```

```
    struct
AlohaCOM
{
    AlohaCOM()  {  CoInitialize(NULL);  }
   ~AlohaCOM()  {  CoUninitialize();  }
} g_AlohaCOM;

    void
guard_( _com_error   e,
        char const * statement = NULL,
        char const * file      = NULL,
        int          line      = 0 )
{
    stringstream out;

    if (file)
        out << file << "(" << line << ") : ";

    out << "COM error 0x";
    out.fill('0');
    out << std::setw(8) << std::hex << e.Error();

    if (statement)
        out << " executing: " << statement;

    out << "\n\t" << e.Description();
    out << "\n\t" << e.ErrorMessage();
    cout << out.str() << endl;
    OutputDebugStringA(out.str().c_str());
    OutputDebugStringA("\n");
}

#ifdef _DEBUG
#   define ELIDE_(x) x
#else
#   define ELIDE_(x) // x
#endif

#define GUARD(statement) do { HRESULT hr = (statement);  \
    if (FAILED(hr))  {                                   \
        guard_(hr, #statement, __FILE__, __LINE__);      \
        ELIDE_(__asm { int 3 });                         \
        _com_issue_error(hr); } } while(false)

    inline _bstr_t
formatParseError( IXMLDOMDocument2Ptr &pXML )
{
    MSXML2::IXMLDOMParseErrorPtr pError = pXML->parseError;

    return _bstr_t("At line ") +
           _bstr_t(pError->Getline()) +
           _bstr_t("\n") +
           _bstr_t(pError->Getreason());
}

CustomerAddress::CustomerAddress(CString xml)
    {
    GUARD(m_pXML.CreateInstance(L"Msxml2.DOMDocument.4.0"));
    m_pXML->setProperty("SelectionLanguage", "XPath");
    m_pXML->async = true;

    VARIANT_BOOL parsedOkay = m_pXML->loadXML(_bstr_t(xml));

    if (!parsedOkay)
```

```
        {
        cout << formatParseError(m_pXML);
        ELIDE_(__asm { int 3 });
        }
    }

    CString
CustomerAddress::get(CString field)
{
    _bstr_t xPath = "/users/user[1]/" + field + "/text()";
    XMLNodePtr_t pField = m_pXML->selectSingleNode(xPath);

    return pField.GetInterfacePtr()?
                pField->text.operator LPTSTR(): TEXT("");
}

    void
CustomerAddress::set(CString field, TCHAR const * nuText)
{
    _bstr_t xPath = "/users/user[1]/" + field;
    XMLNodePtr_t pField = m_pXML->selectSingleNode(xPath);
    pField->text = nuText;
}

    class
Field
{
  public:
    Field(char const * name, UINT id):
        m_name(name),
        m_id(id)
            {}

    virtual void dialogFromXML( ProjectDlg      &aDlg,
                                CustomerAddress &aCA );

    virtual void dialogToXML( ProjectDlg      &aDlg,
                              CustomerAddress &aCA );

  protected:
    CString const m_name;
    UINT    const m_id;
};

    class
ComboField: public Field
{
  public:
    ComboField(char const * name, UINT id):
        Field(name, id)
            {}

    void dialogFromXML( ProjectDlg      &aDlg,
                        CustomerAddress &aCA );
};

    CStrings_t
split(CString str, char delim)
{
    CStrings_t list;

    for (;;)
        {
```

```
        int at = str.Find(delim);
        if (at == -1)  break;
        list.push_back(str.Mid(0, at));
        str = str.Mid(at + 1);
        }
    if (str.GetLength())
        list.push_back(str);

    return list;
}

    Cstrings_t
CustomerAddress::getStateList()
{
    _bstr_t xPath = "/users/user[1]/state/@options";
    XMLNodePtr_t pField = m_pXML->selectSingleNode(xPath);
    return pField->text.operator LPCSTR();
}

    void
ComboField::dialogFromXML( ProjectDlg      &aDlg,
                           CustomerAddress &aCA )
{
    CStrings_t states = aCA.getStateList();
    CComboBox aBox(aDlg.GetDlgItem(m_id));

    for ( CStrings_t::iterator it(states.begin());
          it != states.end();  ++it )
        {
        int at = aBox.AddString(*it);
        assert(at > -1);
        }

    CString selectedState = aCA.get(m_name);
    aBox.SelectString(-1, selectedState);
}


    struct  //  a "convenience class"
FieldPtr_t: std::auto_ptr<Field>
{
    FieldPtr_t(Field *p):
        std::auto_ptr<Field>(p) {}

};  //  explicit is defeated!!

    FieldPtr_t
fields[] = {
    new Field( "first_name" , IDC_EDIT_FIRST_NAME ),
    new Field( "last_name"  , IDC_EDIT_LAST_NAME  ),
    new Field( "address_1"  , IDC_EDIT_ADDRESS_1  ),
    new Field( "address_2"  , IDC_EDIT_ADDRESS_2  ),
    new Field( "city"       , IDC_EDIT_CITY       ),
    new ComboField( "state" , IDC_COMBO_STATE     ),
    new Field( "zip"        , IDC_EDIT_ZIP        ),
    };

    void
Field::dialogFromXML( ProjectDlg      &aDlg,
                      CustomerAddress &aCA )
    {
    aDlg.SetDlgItemText(m_id, aCA.get(m_name));
    }
```

```
    void
Field::dialogToXML(ProjectDlg &aDlg, CustomerAddress &aCA)
    {
    aCA.set(m_name, aDlg.getText(m_id));
    }

    void
ProjectDlg::transferXML
    (
    void (Field::*transfer)( ProjectDlg      &aDlg,
                             CustomerAddress &aCA )
    )
{
    for ( int x(0);
          x < sizeof fields / sizeof fields[0];
          ++x )
        {
        Field * pField = fields[x].get();
        (pField->*transfer)(*this, m_aCA);
        }
}

    void
ProjectDlg::saveXML()
{
    transferXML(&Field::dialogToXML);
}

    LRESULT
ProjectDlg::OnInitDialog(UINT, WPARAM, LPARAM, BOOL &)
{
    transferXML(&Field::dialogFromXML);
    return 0;
}

    LRESULT
ProjectDlg::OnClose(UINT uCode, int nID, HWND, int)
{
    if (m_bModal)
        EndDialog(nID);
    else
        PostMessage(WM_QUIT);

    return 0;
}

    CString
ProjectDlg::getText(UINT ID) const
{
    CString text;
    GetDlgItemText(ID, text);
    return text;
}

    CString
readFile(CString fileName)
{
    CString contents;
    std::ifstream in(fileName);
    char ch;
    while (in.get(ch))  contents += ch;
```

```
    return contents;
}

    void
runApp(CString fileName)
{
    CString xml = readFile(fileName);
    ProjectDlg aDlg(xml);
    aDlg.DoModal();
}

    int
main(int argc, char **argv)
{

    if (argc != 2 || argc > 4)
        {
        std::cerr << "Arguments: fileName.xml";
        ELIDE_(std::cerr << " or --test");
        std::cerr << endl;
        return 1;
        }

    try {
        CString arg(argv[1]);

        ELIDE_(
            if (arg == "--test")
                return ! runTests()
            );

        runApp(arg);
        }
    catch(_com_error &e)
        {
        guard_(e);
        ELIDE_(__asm { int 3 });
        return 1;
        }

    return 0;

}
```

The `main()` has more statements, all written via the miracle of Test-Free Programming. They check for a command line argument, then either run tests (only available in Debug Mode) or display a given XML file. It currently may contain only one record; we fix this next.

Multiple Customers

To force `CustomerAddress` to handle more than one record, I changed the test resource XML to look like this (with the payload text in **`bold`** for clarity):

```
<users>
  <user>
    <first_name>Ignatz</first_name>
    <last_name>Mouse</last_name>
    <address_1>Heppy Land</address_1>
    <address_2></address_2>
    <city>Coconino County</city>
```

```
    <state options='AK,AL,AR,AZ,'>AZ</state>
    <zip>86351</zip>
  </user>
  <user>
    <first_name>Krazy</first_name>
    <last_name>Kat</last_name>
    <address_1>Heppy Land</address_1>
    <address_2></address_2>
    <city>Coconino County</city>
    <state options='AK,AL,AR,AZ,'>AZ</state>
    <zip>86351</zip>
  </user>
  <user>
    <first_name>Offisa</first_name>
    <last_name>Pup</last_name>
    <address_1>Heppy Land</address_1>
    <address_2></address_2>
    <city>Coconino County</city>
    <state options='AK,AL,AR,AZ,'>AZ</state>
    <zip>86351</zip>
  </user>
</users>
```

The state codes, "`AK,AL,AR,AZ,`" duplicate in each record. A more complete Representation Layer should easily fix this, without influencing the GUI. Removing this minor duplication won't teach us much, so we leave it there.

Our next step is a similar upgrade, behind the layer interface, to upgrade `CustomerAddress` to ignore all but the first record.

The previous chapter used XPath queries to find one data record. So a query of `/users/user/first_name/text()` assumed only one record matched. Adding more `<user>` records breaks that assumption.

To permit the data to upgrade, without changing the tests or the interfaces, the `CustomerAddress` methods now use XPath with a hack: "`[1]`". No test yet forces it to select anything but the first record:

```
    CString
CustomerAddress::get(CString field)
{
    _bstr_t xPath = "/users/user[1]/" + field + "/text()";
    XMLNodePtr_t pField = m_pXML->selectSingleNode(xPath);

    return pField.GetInterfacePtr()?
               pField->text.operator LPTSTR(): TEXT("");
}

    void
CustomerAddress::set(CString field, TCHAR const * nuText)
{
    _bstr_t xPath = "/users/user[1]/" + field;
    XMLNodePtr_t pField = m_pXML->selectSingleNode(xPath);
    pField->text = nuText;
}

    CString
CustomerAddress::getStateList()
{
    _bstr_t xPath = "/users/user[1]/state/@options";
    XMLNodePtr_t pField = m_pXML->selectSingleNode(xPath);
    return pField->text.operator LPCSTR();
```

```
}
```

Come to think of it, if I had continued to merge duplication before adding more customer address records, those three sub-strings would have been in one place, for an even quicker hack. We'll do that now, before forcing `CustomerAddress` to select different customers by changing the `1` inside that `[1]` record index:

```
    XMLNodePtr_t
CustomerAddress::getNode(CString xPath)
{
    _bstr_t bPath = "/users/user[1]/" + xPath;
    return m_pXML->selectSingleNode(bPath);
}

    CString
CustomerAddress::getStateList()
{
    XMLNodePtr_t pField = getNode("state/@options");
    return pField->text.operator LPCSTR();
}

    CString
CustomerAddress::get(CString field)
{
    XMLNodePtr_t pField = getNode(field + "/text()");

    return pField.GetInterfacePtr()?
               pField->text.operator LPTSTR(): TEXT("");
}

    void
CustomerAddress::set(CString field, TCHAR const * nuText)
{
    XMLNodePtr_t pField = getNode(field);
    pField->text = nuText;
}
```

To teach `CustomerAddress` to vary the `[1]` index, we could write a test on `ProjectDlg` that requests it to display "`Krazy Kat`" instead of "`Ignatz Mouse`". Sometimes a test will request Object A to display a behavior that Object B must change to support.

Instead, we speculate that `ProjectDlg` will soon need that new ability, and add it directly to `CustomerAddress`:

```
TEST_(TestCase, setRecordIndex)
{
    CustomerAddress aCA(xml);

    aCA.setRecordIndex(1);
    CPPUNIT_ASSERT_EQUAL("Ignatz", aCA.get("first_name"));

    aCA.setRecordIndex(2);
    CPPUNIT_ASSERT_EQUAL("Krazy",  aCA.get("first_name"));

    aCA.setRecordIndex(3);
    CPPUNIT_ASSERT_EQUAL("Offisa", aCA.get("first_name"));
}
```

Note that interface exposes XPath's `1`-based indexing. `Ignatz`'s index is not `0`. And because we don't honestly yet know the circumstances that will change the record index, we don't address out-of-bounds situations yet.

The new infrastructure to pass the test requires a private index member inside `CustomerAddress`:

```
        class
    CustomerAddress
    {
      public:
        void    setRecordIndex(int idx)  {  m_idx = idx;  }

      private:
        int m_idx;
…
    };

    CustomerAddress::CustomerAddress(CString xml):
            m_idx(1)
        {
…
        }
```

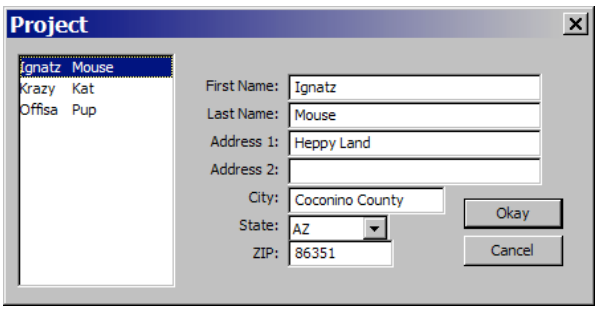
And the new improved `getNode()` uses this index:

```
        XMLNodePtr_t
    CustomerAddress::getNode(CString xPath)
    {
        stringstream sPath;
        sPath << "/users/user[" << m_idx << "]/";
        _bstr_t bPath = sPath.str().c_str() + xPath;
        return m_pXML->selectSingleNode(bPath);
    }
```

`CustomerAddress` is ready for the list box on the side of our dialog, but `ProjectDlg` is not. The next few changes must fix its simplest problems first.

Saving Data

After a few more features, our dialog will look like this:



The "Okay" button now saves changes. This should raise some GUI usability questions:

- If the current record changed, the "Cancel" button must defend the change
- `CustomerAddress` must detect changes at `Set()` time, and remember them

- When we add the ability to switch the GUI to different records, switching away from a changed record should not discard the changes.

Our project will not address most of those problems. The test-first techniques covered so far should easily enforce their fixes. A series of test cases can set up each scenario and detect the correct responses. This narrative will touch a few problems, such as detecting changes at `set()` time:

```
TEST_(TestCase, getModified)
{
    CustomerAddress aCA(xml);
    CPPUNIT_ASSERT( ! aCA.getModified() );
    aCA.set("first_name", "Pogo");
    CPPUNIT_ASSERT( aCA.getModified() );

  //  after we learn to Save, test that Saving resets m_modified

}
```

Note that, in C++, comments may refer to private variables, such as `m_modified`. That's risky. The variable might metamorphosize, while the comment still passes all automated tests. If we treat all comments with suspicion, then a good use for comments is to raise suspicion. In this case, we suspect we might forget to eventually write another test. But the comment cannot document with assurance that `m_modified` still exists.

To pass the previous test, add a private `m_modified` member, and its Getter, then Set it to true when any XML data field changes its value:

```
    class
CustomerAddress
{
  public:
…
    bool    getModified()  {  return m_modified;  }

  private:
…
    bool m_modified;
};
…
CustomerAddress::CustomerAddress(CString xml):
        m_idx(1),
        m_modified(false)
    {
…
    }
…
    void
CustomerAddress::set(CString field, TCHAR const * nuText)
{
    CString oldText = get(field);

    if (oldText != nuText)
        {
        XMLNodePtr_t pField = getNode(field);
        pField->text = nuText;
        m_modified = true;
```

```
        }
}
```

Now we demonstrate that the Okay button saves. A new test case gives the `ProjectDlg` a file name, changes an address, clicks the Okay button, and checks that the new address value went into the file. Note the test `removes` any previous incarnation of this file:

```
TEST_(TestDialog, OnClose)
{
    CString fileName("test.xml");
    remove(fileName);  //  might remain from last test run

    m_aDlg.setFileName(fileName);

    CString address_2 ("Lithoid Mittens");
    m_aDlg.SetDlgItemText(IDC_EDIT_ADDRESS_2, address_2);
    CButton Save(m_aDlg.GetDlgItem(IDOK));
    Save.SendMessage(BM_CLICK);
    CString nuXML(readFile(fileName));
    CustomerAddress aCA(nuXML);
    CPPUNIT_ASSERT_EQUAL(address_2, aCA.get("address_2"));
}
```

That test uses `SendMessage()` for *Firm User Simulation* (per page **Error! Bookmark not defined.**). When `BM_CLICK` simulates a real user clicking the Save button, we rely on MS Windows to behave similar to real user input.

While addressing these usability points, we can ensure that closing our dialog saves its data:

```
    LRESULT
ProjectDlg::OnClose(UINT uCode, int nID, HWND, int)
{
    saveXML();

    if (nID == IDOK && m_aCA.getModified())
        {
        m_aCA.getXML()->save(_variant_t(m_fileName));
        }
    if (m_bModal)
        EndDialog(nID);
    else
        PostMessage(WM_QUIT);

    return 0;

}
```

The implementation passes the test by saving the dialog's fields back into the XML, then calling MSXML's built-in `save()` method.

Note that's the test that forced `getXML()` to exist (see the complete source code listing a couple pages back). One should not use test-first directly to produce functions with trivial implementations, such as accessors. Only add these to fulfill other higher-level tests.

List Box Population

We need a list box to store the persisted records. Author one, up the left side:

```
#define IDC_EDIT_ZIP                    1007
#define IDC_LIST_CUSTOMERS              1008

...
IDD_PROJECT DIALOGEX 0, 0, 294, 122

...
BEGIN
    LISTBOX         IDC_LIST_CUSTOMERS, 5,7,79,108,
                            LBS_SORT | LBS_NOINTEGRALHEIGHT |
                            WS_VSCROLL | WS_TABSTOP
    RTEXT           "First Name:",IDC_STATIC,92,17,45,10

...
END
```

Notice we cleaned up a few esthetics, and we put the new `LISTBOX` first, not last, in the resource notation, because it's upstream from the others in <Tab> order.

To fill the list box, a new test creates a method `CustomerAddress::getAllNames()`, and its implementation returns an STL list of them:

```
    TEST_(TestCase, getAllNames)
    {
        CustomerAddress aCA(xml);
        CStrings_t allNames = aCA.getAllNames();
        CStrings_t::const_iterator it = allNames.begin();
        CPPUNIT_ASSERT_EQUAL( "Ignatz\tMouse", *it );  ++it;
        CPPUNIT_ASSERT_EQUAL( "Krazy\tKat",    *it );  ++it;
        CPPUNIT_ASSERT_EQUAL( "Offisa\tPup",   *it );  ++it;
        CPPUNIT_ASSERT(allNames.end() == it);
    }
...
        CStrings_t
    CustomerAddress::getAllNames()
    {
        CStrings_t allNames;
        typedef MSXML2::IXMLDOMNodeListPtr pList_t;

        pList_t first_names =
            m_pXML->selectNodes("/users/user/first_name/text()");

        pList_t last_names =
            m_pXML->selectNodes("/users/user/last_name/text()");

        for (long x = 0;  x < first_names->length; ++x)
            {
            typedef MSXML2::IXMLDOMNodePtr pNode_t;
            pNode_t first_name_node = first_names->item[x];
            pNode_t  last_name_node =  last_names->item[x];

         CString name = first_name_node->text.operator LPCSTR ();

            name += "\t";
            name += last_name_node->text.operator LPCSTR ();
            allNames.push_back(name);
            }

        return allNames;
    }
```

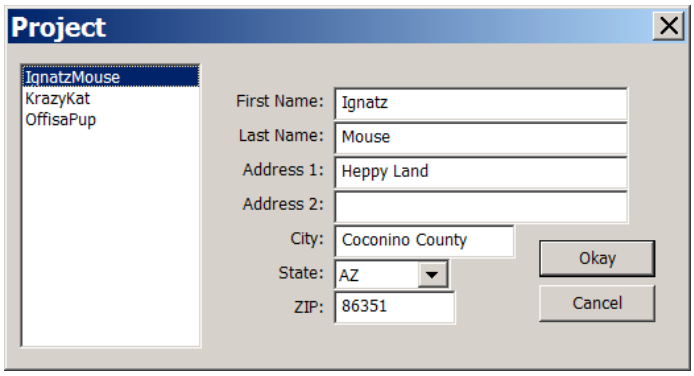
That long method inspires nostalgia for three-line functions. Our excuse is the method does things this program has never done before.

The test case demands `.getAllNames()` to return a collection of names, in their native XML document order, with tab characters `\t` between first and last names. We will need the tabs soon in the list box. The new code queries our database twice, iterates through both `NodeLists` at the same time, concatenates the name parts, and returns the collection.

Now force new code to pour the list of names into the list box, with a test that examines the list box contents via brute-force:

```
    TEST_(TestDialog, getAllNames)
    {
        CListBox aList = m_aDlg.GetDlgItem(IDC_LIST_CUSTOMERS);
        CString name;
        aList.GetText(0, name);
        CPPUNIT_ASSERT_EQUAL( "Ignatz\tMouse", name );

        aList.GetText(1, name);
        CPPUNIT_ASSERT_EQUAL( "Krazy\tKat",    name );

        aList.GetText(2, name);
        CPPUNIT_ASSERT_EQUAL( "Offisa\tPup",   name );

        CPPUNIT_ASSERT_EQUAL(3, aList.GetCount());
        revealFor("phlip");
    }
...
        LRESULT
    ProjectDlg::OnInitDialog(UINT, WPARAM, LPARAM, BOOL &)
    {
        transferXML(&Field::dialogFromXML);
        CStrings_t allNames = m_aCA.getAllNames();
        CListBox   aList    = GetDlgItem(IDC_LIST_CUSTOMERS);

        for ( CStrings_t::iterator it(allNames.begin());
              it != allNames.end();  ++it )
            {
            int at = aList.AddString(*it);
            assert(at > -1);
            }
        aList.SetCurSel(0);
        return 0;
    }
```

As usual, new behavior accumulates in `OnInitDialog()` before it finds a better place. Over time that function grows longer and shorter. Here's the visual inspection:



Each `first_name` and `last_name` has a tab `\t` between them, and the list box displays as them as nothing. We want the names to line up in columns like this:

```
Ignatz  Mouse
Krazy   Kat
Offisa  Pup
```

That's both cute and usable. Our list shall display a first name of "Ignatz Mouse" distinct from a first name "Ignatz" with a last name "Mouse".

To create columns, we pick one of a few ways MS Windows permits developers to get inside a control and assist its `Paint()` event. We won't use `PreTranslateMessage()`, or subclass the window. Those techniques are fragile, so Windows provides a more direct system to override one aspect of a control's repaint event.

Many controls provide an "owner draw" system for some component of their display. In MS Windows, some controls let us set their style bit to an `*_OWNERDRAW*` flag. Then they send `WM_DRAWITEM` events to their parent window, and we write code to handle them.

To test-first a low-level `Paint()` event handler, we need to pass a handle to a Mock Graphics object into a mock `Paint()` events. That requires research to learn what the native mock `Paint()` event looks like, and then research how to capture the graphic commands.

Mock Your GUI Toolkit

Each `WM_DRAWITEM` message passes to your callback the address of a `DRAWITEMSTRUCT` structure, telling you which item to draw and how to emphasize it. To test a handler for this message, we must fake the contents of the structure.

Your Microsoft Developer's Network documentation most likely does not tell you how to fake `DRAWITEMSTRUCT` contents. I learned how by turning on the list box's `LBS_OWNERDRAWFIXED` bit, implementing `OnDrawItem()` as a stub, and forcing the system to call it. Then some trace statements print out its secrets:

```
        LISTBOX         IDC_LIST_CUSTOMERS, 5,7,79,108,
                                LBS_SORT | LBS_NOINTEGRALHEIGHT |
                                WS_VSCROLL | WS_TABSTOP |
                                LBS_OWNERDRAWFIXED | LBS_HASSTRINGS
        RTEXT           "First Name:",IDC_STATIC,92,17,45,10
…
        BEGIN_MSG_MAP(ProjectDlg)
            MESSAGE_HANDLER(WM_INITDIALOG, OnInitDialog)
            MESSAGE_HANDLER(WM_DRAWITEM,   OnDrawItem)
            COMMAND_ID_HANDLER(IDOK,     OnClose)
            COMMAND_ID_HANDLER(IDCANCEL, OnClose)
        END_MSG_MAP()

        LRESULT OnDrawItem(UINT, WPARAM, LPARAM lParam, BOOL &);
…
        LRESULT
    ProjectDlg::OnDrawItem(UINT, WPARAM, LPARAM lParam, BOOL &)
    {
        DRAWITEMSTRUCT &aDrawItem =
            *reinterpret_cast<LPDRAWITEMSTRUCT>(lParam);

        cout << aDrawItem.CtlType << endl;
        cout << aDrawItem.CtlID << endl;
        cout << aDrawItem.itemID << endl;
```

```
…
    }
```

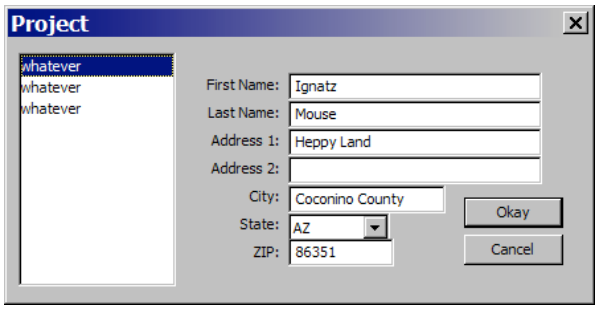
That printed out a list of numbers. Looking up their codes in the header files shows us how to begin a *Temporary Interactive Test*. We construct `aDrawItem`, and send it in a `WM_DRAWITEM` message to our dialog:

```
TEST_(TestDialog, WM_DRAWITEM)
{
    CListBox aList = m_aDlg.GetDlgItem(IDC_LIST_CUSTOMERS);

    DWORD style = aList.GetStyle();
    CPPUNIT_ASSERT(style & LBS_OWNERDRAWFIXED);
    CPPUNIT_ASSERT(style & LBS_HASSTRINGS);

    DRAWITEMSTRUCT aDrawItem = { ODT_LISTBOX };
    aDrawItem.CtlID      = IDC_LIST_CUSTOMERS;
    aDrawItem.itemID     = 0;
    aDrawItem.itemAction = ODA_FOCUS;
    aDrawItem.itemState  = ODS_SELECTED;
    aDrawItem.hwndItem   = aList;
    CDC aDC = aList.GetDC();  //  destructor calls DeleteDC()
    aDrawItem.hDC = aDC;
    aList.GetItemRect(0, &aDrawItem.rcItem);

    m_aDlg.SendMessage(
      WM_DRAWITEM, 0, reinterpret_cast<LPARAM>(&aDrawItem) );
}
```

That test has no assertions after it. We add `revealFor()` to the test, then concoct a deliberately bogus implementation of `OnDrawItem()`. The source code to pass the test will soon force the test to improve:

```
    LRESULT
ProjectDlg::OnDrawItem(UINT, WPARAM, LPARAM lParam, BOOL &)
{
    DRAWITEMSTRUCT &aDrawItem =
        *reinterpret_cast<LPDRAWITEMSTRUCT>(lParam);

    CDCHandle aDC(aDrawItem.hDC);

    BOOL worked = aDC.TextOut
                            (
                            aDrawItem.rcItem.left,
                            aDrawItem.rcItem.top,
                            "whatever"
                            );
    assert(worked);
    return 0;
}
```

Because our test is not yet aggressive enough, that code passes it.
The resulting window looks like this:



In simple data entry applications, when you go to supply your list boxes' `OnDrawItem()` behaviors, I give you permission to type in the obvious implementation, and not to frantically test it. I want to see if I can force lies out of my implementation. That will lead to test fixtures that assist developing raw graphics much more complex than list box text. Other Case Studies have shown how aggressively growing test fixtures now permits high velocity later.

GDI MetaFiles

To test that `OnDrawItem()` drew something, we must pass in a Mock Graphics device context capable of recording each "Graphics Device Interface" call. MS Windows's GDI layer provides a system called "MetaFiles" to retain graphic commands into files. Our strategy is to create a MetaFile-based device context, put it into a `DRAWITEMSTRUCT`, and pass this into the `WM_DRAWITEM` handler. After it returns, the test case will `Close()` the MetaFile-based device context and extract a `HMETAFILE` handle to the new file. The test case then will parse this file to verify it contains the correct GDI commands.

Remember; if you use a different graphics layer, and you don't want bugs in your customized controls, you must perform the same kind of research to make that graphics layer testable. GDI MetaFiles provide an out-of-the-box implementation of Mock Graphics (per page 33). Your project might not have a system like MetaFiles, or it may have one that you decline to use, so you might have to build a Mock Graphics system yourself by hand. Soon each GUI Toolkit's community may support a Free Software solution here.

The more our industry automates tests, the easier each new test will become. But our first attempts might appear as ugly as this prototypical Log String test:

```
TEST_(TestDialog, WM_DRAWITEM)
{
    CListBox aList = m_aDlg.GetDlgItem(IDC_LIST_CUSTOMERS);

    DWORD style = aList.GetStyle();
    CPPUNIT_ASSERT(style & LBS_OWNERDRAWFIXED);
    CPPUNIT_ASSERT(style & LBS_HASSTRINGS);

    CDC aDC = aList.GetDC();

  //  calculate the correct size for the MetaFile

    int mmWidth      = aDC.GetDeviceCaps(HORZSIZE);
    int mmHeight     = aDC.GetDeviceCaps(VERTSIZE);
    int pixelsWidth  = aDC.GetDeviceCaps(HORZRES);
    int pixelsHeight = aDC.GetDeviceCaps(VERTRES);
    int xMmPerPixel  = (mmWidth  * 100) / pixelsWidth;
    int yMmPerPixel  = (mmHeight * 100) / pixelsHeight;
```

```
    RECT rc;
    aList.GetItemRect(0, &rc);
    rc.left   *= xMmPerPixel;
    rc.top    *= yMmPerPixel;
    rc.right  *= xMmPerPixel;
    rc.bottom *= yMmPerPixel;

  //  create a MetaFile-based device context

    CEnhMetaFileDC aDCmeta;
    aDCmeta.Create(aDC, "sample.emf", &rc, "test");

  //  fake a WM_DRAWITEM message

    DRAWITEMSTRUCT aDrawItem = { ODT_LISTBOX };
    aDrawItem.CtlID      = IDC_LIST_CUSTOMERS;
    aDrawItem.itemID     = 0;
    aDrawItem.itemAction = 0;
    aDrawItem.itemState  = 0;
    aDrawItem.hwndItem   = aList;
    aDrawItem.hDC        = aDCmeta;
    aDrawItem.rcItem     = rc;
    LPARAM param = reinterpret_cast<LPARAM>(&aDrawItem);
    m_aDlg.SendMessage(WM_DRAWITEM, 0, param);

  //  Closing the DC returns a HMETAFILE

    CEnhMetaFile metaFile(aDCmeta.Close());

  //  inspect the MetaFile for the correct record

    std::wstringstream payload;

    BOOL worked = EnumEnhMetaFile
                  (
                  NULL,
                  metaFile,
                  metaFileRecordFunction,
                  &payload,
                  &aDrawItem.rcItem
                  );

    CPPUNIT_ASSERT(worked);
    CPPUNIT_ASSERT(L"Ignatz\tMouse" == payload.str());
}
```

The SDK function `EnumEnhMetaFile()` requires we write a callback, `metaFileRecord-Function()`, to interpret each record in the MetaFile. Our code is about to get even uglier.

Many MS Windows enumeration functions, such as `EnumEnhMetaFile()`, express dynamic types within a static typed language. That forces copious typecasts between `void` pointers and typed ones. If our Sane Subset refuses to consider C-style typecasts (`(char const *)rec`), the best implementation of `metaFileRecord()` we can hope for is this:

```
    int CALLBACK
metaFileRecordFunction
        (
        HDC,
        HANDLETABLE FAR *,
        CONST ENHMETARECORD *rec,
        int,
```

```
        LPARAM param
        )
{
    switch (rec->iType)
        {
        case EMR_HEADER:
        case EMR_EOF:
            break;

        case EMR_EXTTEXTOUTW:
            {
            EMREXTTEXTOUTW const &aTextOut =
     *reinterpret_cast<EMREXTTEXTOUTW const *>(rec);

            std::wstringstream &payload =
     *reinterpret_cast<std::wstringstream*>(param);

            char const * p =
                reinterpret_cast<char const *>(rec);


            p += aTextOut.emrtext.offString;  //  offset in bytes

            wchar_t const * text =
                reinterpret_cast<wchar_t const *> (p);

            payload.write(text, aTextOut.emrtext.nChars);
            }
            break;

        default:  assert(false);
        }
    return true;
}
```

Those big long statements with `reinterpret_cast<char const *>(rec)` are good because they advertise their risks. When complex data structures squeeze through these interfaces, they lose their type information. Both the MetaFile's `EMR` records and the test case's `wstringstream` object pass by typeless address, so they both need ugly `reinterpret_cast<>` calls to recover their types. Low-level systems require many hacks to achieve the Execute Around Pattern.

The Windows SDK can bond to any language, so it can't use language-specific typesafe declarations. Our C++ style guidelines require fragile code to *appear* fragile, with `reinterpret_cast<>` spelled out. That reminds us to treat these statements very delicately. "`reinterpret`" means the compiler coerces one type to another without any logical conversion. This only works when programmers know both types match. If they don't, the compiler cannot warn, and anything could happen.

To complicate this situation more, some output types don't match the production code's input types. These statements extract the 16-bit text from a `EMREXTTEXTOUTW` record. (The Localization project, on page 255, will explore a few more such string width issues.) When our production code called `TextOut()`, it really called `TextOutA()`. That "ANSI" version of the function took 8-bit `char` text. Something in the OS expanded it to 16-bit `wchar_t` text.

My operating system, Windows XP, derives from Windows NT. It will never run this program and return `EMREXTTEXTOUTA` records containing 8-bit text. OSs derived from NT's arch-enemy, Windows 95, might return 8-bit text.

Platforms in the NT lineage all internally convert strings into their wide, UCS versions. On NT, `TextOutA()` widened its string argument and passed it to `TextOutW()`. That created an `EMREXTTEXTOUTW` record and pushed this into the `hDC` device context. That context is usually a

window, but our test provided a MetaFile destination. MetaFile readers traverse these retained records, which thought they would go to a window.

Inside `metaFileRecordFunction()`, the `char` pointer `p` points to that widened string inside the `EMREXTTEXTOUTW` record. Then `text` points to the same place but through a pointer to wide characters, and `payload.write()` pushes that wide string into our test's Log String—a wide `wstringstream`. I had to use `.write()`, not `operator<<`, because `text` points to a string without a `L'\0'` sentinel at its end.

All that infrastructure climaxes in this simple line:

```
    CPPUNIT_ASSERT(L"Ignatz\tMouse" == payload.str());
```

Our wide strings could not use `CPPUNIT_ASSERT_EQUAL()`, because that expects 8 bits. Eventually a wide version, `CPPUNIT_ASSERT_EQUAL_W()`, will cover these `wchar_t` situations.

Oh, by the way, that assertion fails. The Bootstrapping phase is over, we successfully detected the lie we planted in the production code, and test-first begins here.

Log String Test

That code had a very nice side effect. It left behind a file called `sample.emf`. To view it, navigate Windows Explorer to it and double-click on it, or run `MSPaint.exe` and open it:

The extra black appeared because the list box paints itself white before our function draws that text. Our EMF file contains only the text command, because our only drawing command, `TextOut()`, pushed in enough background color to back-fill the text it drew. Then `MSPaint.exe` sets its own background to black before playing the MetaFile.

Examining this file provides an excellent double-check. We have discovered yet another way to treat high-bandwidth graphical output like a lowly spigot, to improve the odds that display bugs get caught early (including the one we deliberately planted). For example, if the text's coordinates were wrong, our test don't check for that situation, and would pass. If the EMF file revealed such problems, we would see them and increase test coverage in that direction.

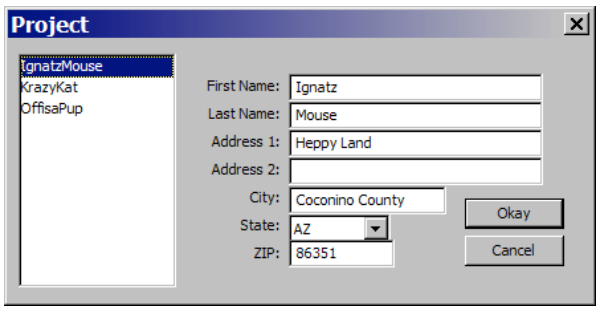
The Mock Graphics technique relies on Log Strings, which can be fragile. Because MS Windows provides the Mock GDI layer for us, these industrial-strength features make the technique more robust.

To help that EMF file not display "`whatever`", and to pass our current test, new production code extracts the string ("`Ignatz\tMouse`") from the list box's internal database and displays it:

```
    LRESULT
ProjectDlg::OnDrawItem(UINT, WPARAM, LPARAM lParam, BOOL &)
{
    DRAWITEMSTRUCT &aDrawItem =
        *reinterpret_cast<LPDRAWITEMSTRUCT>(lParam);

    CDCHandle aDC(aDrawItem.hDC);
    CListBox aList = aDrawItem.hwndItem;
    CString line;
    aList.GetText(aDrawItem.itemID, line);

    BOOL worked = aDC.TextOut
                      (
                      aDrawItem.rcItem.left,
```

```
                        aDrawItem.rcItem.top,
                        line
                        );
    assert(worked);
    return 0;
}
```

The light may be green, but the trap is not yet clean. Add `revealFor()` to any test in the `TestDialog` suite, and manually inspect:



The first and last names in the list still cram together. We have not yet applied the reason for our `LBS_OWNERDRAWFIXED` setting—expressing `\t` characters in the list. And we regressed a little. Clicking on items in the list does not render their selection emphases. Users would not know which item was selected.

To fix this, search the Internet for `LBS_OWNERDRAWFIXED`, and download sample projects from WTL web sites. Then compile and run them, put breakpoints inside them, and examine their source code in action. (But don't just copy and paste their code into your project!)

Our strategy now is to clone the test, and name the copy after the `ODS_SELECTED` bit it will eventually use. Then make sure the unmodified clone passes, and move much of the `CEnhMetaFile` stuff into test fixtures. This gives two tests that do the same thing.

Then the cloned test will turn on the `ODS_SELECTED` bit and declare its target item as selected, but it won't change its assertion. We don't know yet what the new MetaFile will contain when the item is selected.

Then we will change our source code to match those sample WTL projects, make the test fail, reveal the window, *Temporarily Interactively Test* that the behavior is correct, and then change the assertion to match what the code is now doing.

> This is not cheating on test-first; this is research. Configuring test fixtures to accept behaviors confirmed via visual inspections leads to new test cases that reuse these test fixtures proactively, decreasing the need for *Temporary Visual Inspections*.

The new test fixtures have obvious implementations, so here's the new test alone:

```
TEST_(TestDialog, WM_DRAWITEM_ODS_SELECTED)
{
    CListBox aList = m_aDlg.GetDlgItem(IDC_LIST_CUSTOMERS);
    CDC aDC = aList.GetDC();
    RECT rc(getSizedRectangle(aList));

    CEnhMetaFileDC aDCmeta;
```

```
    aDCmeta.Create(aDC, "sample.emf", &rc, "test");

    DRAWITEMSTRUCT aDrawItem = { ODT_LISTBOX };
    aDrawItem.CtlID      = IDC_LIST_CUSTOMERS;
    aDrawItem.itemID     = 0;
    aDrawItem.itemAction = 0;
    aDrawItem.itemState  = ODS_SELECTED;
    aDrawItem.hwndItem   = aList;
    aDrawItem.hDC        = aDCmeta;
    aDrawItem.rcItem     = rc;
    LPARAM param = reinterpret_cast<LPARAM>(&aDrawItem);

    m_aDlg.SendMessage(WM_DRAWITEM, 0, param);

    std::wstring payload =
        metaFileToString(aDCmeta.Close(), aDrawItem.rcItem);

    CPPUNIT_ASSERT(L"Ignatz\tMouse" == payload);
}
```

According to our plan, it still passes, even though the `.itemState` is now `ODS_SELECTED`.

Peek at some WTL sample code, and upgrade the code to make the test fail. `OnDrawItem()` responds to `ODS_SELECTED` by choosing system colors with "`HIGHLIGHT`" in their names:

```
    LRESULT
ProjectDlg::OnDrawItem(UINT, WPARAM, LPARAM lParam, BOOL &)
{
    DRAWITEMSTRUCT &aDrawItem =
        *reinterpret_cast<LPDRAWITEMSTRUCT>(lParam);

    CDCHandle aDC(aDrawItem.hDC);

    CListBox aList = aDrawItem.hwndItem;
    CString line;
    aList.GetText(aDrawItem.itemID, line);

    bool selected = aDrawItem.itemState & ODS_SELECTED;

    aDC.FillRect
            (
            &aDrawItem.rcItem,
            selected? COLOR_HIGHLIGHT: COLOR_WINDOW
            );

    DWORD color      = GetSysColor( selected?
                           COLOR_HIGHLIGHTTEXT:
                           COLOR_WINDOWTEXT );

    DWORD background = GetSysColor( selected?
                           COLOR_HIGHLIGHT:
                           COLOR_WINDOW );

    COLORREF crOldText = aDC.SetTextColor(color);
    COLORREF crOldBack = aDC.SetBkColor(background);

    BOOL worked = aDC.TextOut
                        (
                        aDrawItem.rcItem.left,
                        aDrawItem.rcItem.top,
```

```
                        line
                        );
    assert(worked);
    aDC.SetTextColor(crOldText);
    aDC.SetBkColor(crOldBack);
    return 0;
}
```

That function grew crufty because GDI maintains at least two systems to declare a color; `COLORREF`, which is a 32-bit value containing actual Red, Green, Blue in 8-bit fields, and `GetSysColor()`, which converts system color identifiers into handles to brushes. Your Desktop's Control Panel's Display Properties applet maintains the color scheme that provides actual color values for identifiers like `COLOR_WINDOWTEXT`. Some methods in that function take `COLORREF` values, and some take handles to brushes.

The test successfully fails. The new MetaFile contains records that `metaFileRecord-Function()` cannot yet interpret. Inside that function, the `default: assert(false)` fails.

Inverting test-first doesn't really violate the spirit of TFP. We are still ratcheting the code along, testing frequently, predicting the (general) result of each test run, and minimizing edits between passing tests. The code must tell the test what to assert because I don't know the platform well enough. I cannot predict off the top of my head what records `OnDrawItem()` will draw into the MetaFile.

`metaFileRecordFunction()` must decipher more records and push them into the `wstringstream payload`. I want that string to contain the MetaFile's list of commands, converted into an abbreviated format that lets us see only a few important points of each command. So each test should `expect` something like this:

```
wstring expect =
    L"CREATEBRUSHINDIRECT(0x1),"
    L"SELECTOBJECT(0x1),"
    L"BITBLT(0x0),"
    L"SELECTOBJECT(0x80000000),"
    L"SETTEXTCOLOR(0x0),"
    L"SETBKCOLOR(0xffffff),"
    L"EXTTEXTOUTW(Ignatz\tMouse),"
    L"EXTCREATEFONTINDIRECTW(0x2),"
    L"SELECTOBJECT(0x2),"
    L"SELECTOBJECT(0x8000000d),"
    L"DELETEOBJECT(0x2),"
    L"SETTEXTCOLOR(0x0),"
    L"SETBKCOLOR(0xffffff),";
```

A Log String test with a very long reference string resists test-first. One cannot change an element of the log and accurately predict failure, leading to new code. The next level should be test fixtures that treat the Log String with Fuzzy Matches (from page 37).

Each record in the MetaFile is a command to GDI drivers at the next lower layer. Our function must push each command's name, and one of its command variables, into the `payload` stream. I added a bunch of case statements to `metaFileRecordFunction()`, and they all looked very similar.

After the tests pass, the middle of `metaFileRecordFunction()` looks like this:

```
        case EMR_BITBLT:
            {
            EMRBITBLT const &blit =
                *reinterpret_cast<EMRBITBLT const *>(rec);
```

```
            payload << " BITBLT(0x"
                    << hex << blit.crBkColorSrc << "),";
            }
            break;

        case EMR_CREATEBRUSHINDIRECT:
            {
            EMRCREATEBRUSHINDIRECT const &brush =
         *reinterpret_cast<EMRCREATEBRUSHINDIRECT const *>(rec);

            payload << "CREATEBRUSHINDIRECT(0x"
                    << hex << brush.ihBrush << "),";
            }
            break;

        case EMR_DELETEOBJECT:
            {
            EMRSELECTOBJECT const &select =
         *reinterpret_cast<EMRSELECTOBJECT const *>(rec);

            payload << "DELETEOBJECT(0x"
                    << hex << select.ihObject << "),";
            }
            break;
```

That looks very ugly because for each structure like `EMRSELECTOBJECT`, GDI provides a constant integer with a matching name: `EMR_SELECTOBJECT`. Our test fixture devolved into a huge ladder of case statements handling each record.

Record handlers down-cast records into their target object types using a typecast. C style typecasts, such as `(EMRSELECTOBJECT const *)rec`, are fragile, and MS Windows SDK code needs all the help it can get. Per page 188, we only use elaborate C++ typecasts, such as `reinterpret_cast<EMRSELECTOBJECT const *>(rec)`. That refuses to compile if a sloppy refactor changes its types in certain ways.

These types and typecasts are very long, leading to record handlers that are hard to read and easy to break. Worse, most `case` statements perform the same dumb behavior, with trivial differences between each one.

Faced with ugly but repeating behaviors that differ only by unrefactorable types, C++ gives us two options. We could read Andrei Alexandrescu's excellent book, *Modern C++ Design*, and develop clever, brief, and inscrutable "traits" and "policies". These structures could express differences and parallels between types and integers as intricate clusters of partially specialized templates.

Or we can just write a `#define` to put a bullet through the problem. Everything from the `case` to the `break` goes inside one easy macro:

```
#define EMR_(type_, member_) case EMR_##type_:  {     \
     EMR##type_ const &thing =                        \
         *reinterpret_cast<EMR##type_ const *>(rec);  \
     payload << #type_ "(0x"                          \
             << hex << thing.member_ << "),";         \
     }  break;

        EMR_(BITBLT,              crBkColorSrc);
        EMR_(CREATEBRUSHINDIRECT, ihBrush     );
        EMR_(DELETEOBJECT,        ihObject    );
```

That code is hard to read, hard to change, and easy to use. It leads to this test fixture, with a test using it, and a new assertion:

```
using std::wstring;
using std::wstringstream;
using std::hex;

#define WIDEN_(x_) L##x_
#define WIDEN(x_) WIDEN_(x_)

#define CPPUNIT_ASSERT_EQUAL_W(sample, result)             \
    if ((sample) != (result))  {  std::wstringstream out;   \
     out << WIDEN(__FILE__) << L"(" << __LINE__ << L") : "; \
        out << L#sample << L"(" << (sample) << L") != "; \
        out << L#result << L"(" << (result) << L")";     \
        std::wcout << out.str() << endl;                 \
        OutputDebugStringW(out.str().c_str());           \
        OutputDebugStringW(L"\n");                       \
        all_tests_passed = false;                        \
        __asm { int 3 }  }

    void
interpretTextOut
    (
    wstringstream        &payload,
    EMREXTTEXTOUTW const &aTextOut,
    char const           *p
    )
{
    p += aTextOut.emrtext.offString;

    wchar_t const * text =
        reinterpret_cast<wchar_t const *> (p);

    payload << "EXTTEXTOUTW(";
    payload.write(text, aTextOut.emrtext.nChars);
    payload << "),";
}

    int CALLBACK
metaFileRecordFunction
        (
        HDC,
        HANDLETABLE FAR *,
        CONST ENHMETARECORD *rec,
        int,
        LPARAM param
        )
{

    wstringstream &payload =
        *reinterpret_cast<wstringstream*>(param);

    switch (rec->iType)
        {
        case EMR_HEADER:
        case EMR_EOF:
            break;


#define EMR_(type_, member_) case EMR_##type_:  {    \
     EMR##type_ const &thing =                       \
         *reinterpret_cast<EMR##type_ const *>(rec);  \
```

```
     payload << #type_ "(0x"                          \
             << hex << thing.member_ << "),";         \
     }  break;

        EMR_(BITBLT,                 crBkColorSrc);
        EMR_(CREATEBRUSHINDIRECT,    ihBrush     );
        EMR_(CREATEMONOBRUSH,        ihBrush     );
        EMR_(DELETEOBJECT,           ihObject    );
        EMR_(EXTCREATEFONTINDIRECTW, ihFont      );
        EMR_(SELECTOBJECT,           ihObject    );
        EMR_(SETBKCOLOR  ,           crColor     );
        EMR_(SETTEXTCOLOR,           crColor     );

        case EMR_EXTTEXTOUTW:
            {
            interpretTextOut
                (
                payload,
            *reinterpret_cast<EMREXTTEXTOUTW const *>(rec),
                 reinterpret_cast<char const *>(rec)
                );
            }
            break;

        default:
            //assert(false);
            cout << rec->iType << endl;
        }
    return true;
}

    std::wstring
metaFileToString(CEnhMetaFile metaFile, RECT rc)
{
    wstringstream payload;

    BOOL worked = EnumEnhMetaFile
                    (
                    NULL,
                    metaFile,
                    metaFileRecordFunction,
                    &payload,
                    &rc
                    );

    assert(worked);
    return payload.str();
}

TEST_(TestDialog, WM_DRAWITEM_ODS_SELECTED)
{
    CListBox aList = m_aDlg.GetDlgItem(IDC_LIST_CUSTOMERS);
    CDC aDC = aList.GetDC();
    RECT rc(getSizedRectangle(aList));

    CEnhMetaFileDC aDCmeta;
    aDCmeta.Create(aDC, "sample.emf", &rc, "test");

    DRAWITEMSTRUCT aDrawItem = { ODT_LISTBOX };
    aDrawItem.CtlID      = IDC_LIST_CUSTOMERS;
    aDrawItem.itemID     = 0;
    aDrawItem.itemAction = 0;
    aDrawItem.itemState  = ODS_SELECTED;
```

```
    aDrawItem.hwndItem   = aList;
    aDrawItem.hDC        = aDCmeta;
    aDrawItem.rcItem     = rc;
    LPARAM param = reinterpret_cast<LPARAM>(&aDrawItem);

    m_aDlg.SendMessage(WM_DRAWITEM, 0, param);

    std::wstring payload =
        metaFileToString(aDCmeta.Close(), aDrawItem.rcItem);

    wstring expect =
        L"CREATEBRUSHINDIRECT(0x1),"
        L"SELECTOBJECT(0x1),"
        L"BITBLT(0x0),"
        L"SELECTOBJECT(0x80000000),"
        L"SETTEXTCOLOR(0xffffff),"
        L"SETBKCOLOR(0x6a240a),"
        L"EXTTEXTOUTW(Ignatz\tMouse),"
        L"EXTCREATEFONTINDIRECTW(0x2),"
        L"SELECTOBJECT(0x2),"
        L"SELECTOBJECT(0x8000000d),"
        L"DELETEOBJECT(0x2),"
        L"SETTEXTCOLOR(0x0),"
        L"SETBKCOLOR(0xffffff),";

    CPPUNIT_ASSERT_EQUAL_W(expect, payload);

}
```

Microsoft helped us write a convenient `EMR_()` macro by following the principle "same thing same name". Most of the MetaFile record types, after their `EMR_` prefixes, are named the same as the equivalent message structures—after their `EMR` prefixes. "Token Pasting" is one of the uses of the C++ preprocessor that our Sane Subset permits.

That test can't put a macro around `case EMR_EXTTEXTOUTW`. It's the only `case` that extracts text. The `EMREXTTEXTOUTW` record stores the text at an unknown offset of a variable-length record. A new helper function, `InterpretTextOut()`, fishes the text out. All other record types use our `EMR_()` macro to convert MetaFile commands into strings. As we find other reasons to extract strings, we should merge some of that custom code. For now it must remain ugly.

Bulk Assertions

The string `expect`, in that last test, is very long. If it got longer, it would break the testing guideline, "don't assert matches between strings that are very long." We should not, for example, use our MetaFile device context to test from the top of a huge call tree, full of functions that generate zillions of GDI calls. That would force matches to a very long bulky string, and we would have even more trouble changing tests first to add new features.

One formality remains. All controls in MS Windows reflect their "keyboard focus" status with a dotted rectangle:

That rectangle is distinct from the (more familiar) inverted color rectangle indicating a selection. Some list boxes allow users to select by arrowing to multiple items and tapping the <SpaceBar> on each one. These list boxes must not appear to select items that only have the keyboard focus, as an <Up> or <Down> key navigates toward a selection. Users ought to see

only the dotted outline for the keyboard focus, moving to the item. Then a selected item would invert its colors to reflect its selected status.

To put a dotted rectangle around the keyboard focus, add this code to `OnDrawItem()`:

```
        BOOL worked = aDC.TextOut
                            (
                            aDrawItem.rcItem.left,
                            aDrawItem.rcItem.top,
                            line
                            );
        assert(worked);

        if (aDrawItem.itemAction & ODA_FOCUS)
            aDC.DrawFocusRect(&aDrawItem.rcItem);

        aDC.SetTextColor(crOldText);
```

And here's the test that code requires:

```
    TEST_(TestDialog, WM_DRAWITEM_ODA_FOCUS)
    {
…
        aDrawItem.itemAction = ODA_FOCUS;
        aDrawItem.itemState  = 0;
    …
        wstring expect =
            L"CREATEBRUSHINDIRECT(0x1),"
            L"SELECTOBJECT(0x1),"
            L"BITBLT(0x0),"
            L"SELECTOBJECT(0x80000000),"
            L"SETTEXTCOLOR(0x0),"
            L"SETBKCOLOR(0xffffff),"
            L"EXTTEXTOUTW(Ignatz\tMouse),"
            L"EXTCREATEFONTINDIRECTW(0x2),"
            L"SELECTOBJECT(0x2),"
            L"SELECTOBJECT(0x8000000d),"
            L"DELETEOBJECT(0x2),"
            L"CREATEMONOBRUSH(0x2),"
            L"SELECTOBJECT(0x2),"
            L"BITBLT(0x0),"
            L"SELECTOBJECT(0x2),"
            L"BITBLT(0x0),"
            L"SELECTOBJECT(0x2),"
            L"BITBLT(0x0),"
            L"SELECTOBJECT(0x2),"
            L"BITBLT(0x0),"
            L"SELECTOBJECT(0x80000000),"
            L"SETTEXTCOLOR(0x0),"
            L"SETBKCOLOR(0xffffff),";

        CPPUNIT_ASSERT_EQUAL_W(expect, payload);
    }
```

Our bulk assertion got bulkier. Worse, it contains figures like `0x2`, which break the style principle "use named identifiers".

For our test to associate object indices such as `0x2` with real objects, then pull out their properties and display them in the Log String, we would need to write much more code into our fixture. (Page 326329 begins a more complete Log String system that addresses this problem.)

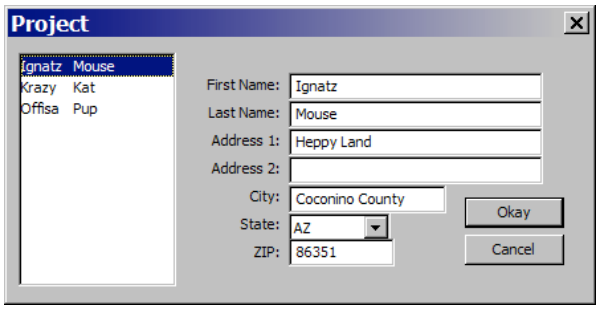
Without those associations, the tests still remain far from the tested code. Reading the above string, we can't see any part of it that says "dotted rectangle". The four new `EMF_BITBLT` messages were doubtless the commands that pushed dotted lines, on each side of the selected item, into video memory. The dottedness may have been an attribute of the "`MONOBRUSH`" thing before them. So the amount of work to spend on this kind of fixture depends on the complexity of the target custom control. A GDI-intense control, such as a Web browser, needs very complex abilities in this fixture.

Run the program and play with it:

The list box now selects and dots correctly.

We have two bugs—oops I mean undeveloped features. The `\t` character still does not expand, and selecting a customer in the list on the left doesn't pump their record into the fields on the right.

To fix the first issue…



…we change the `TextOut()` to a `DrawText()` with `DT_EXPANDTABS`:

```
…
        COLORREF crOldBack = aDC.SetBkColor(background);

           UINT nFormat = DT_LEFT | DT_SINGLELINE |
                       DT_NOPREFIX | DT_VCENTER | DT_EXPANDTABS;

           int height = aDC.DrawText(line, -1, &aDrawItem.rcItem,
                                     nFormat);

        assert(height);

        if (aDrawItem.itemAction & ODA_FOCUS)
            aDC.DrawFocusRect(&aDrawItem.rcItem);
…
```

That change forces us to upgrade the tests. Instead of a fragile bulk assertions, spot check the exact change required:

```
    CPPUNIT_ASSERT(wstring::npos !=
                       payload.find(L"EXTTEXTOUTW(Ignatz)"));

    CPPUNIT_ASSERT(wstring::npos !=
                       payload.find(L"EXTTEXTOUTW(Mouse)"));
```

`aDC.DrawText()` with `DT_EXPANDTABS` processes its input and then calls `TextOut()` on each text segment between tabs, so the Log String now contains separate commands to output `EXTTEXTOUTW(Ignatz)` and `EXTTEXTOUTW(Mouse)`.

A mature TFP project should support a Regular Expression Match (per page 37). Because C++ has no standard regular expressions, and all the alternatives are difficult for me to briefly lecture how to install, we get by with a Fuzzy Match, via `std::wstring::find()` from the C++ Standard Library.

`aDC.DrawText()` also causes a known bug in MetaFiles. That test still writes the `sample.emf` file, like others, but when you view the file, the text is missing.

These tests now adequately constrain our feature. Test-first can now generate more complex `Paint()` events if we write fuzzy matches on our Mock Graphics Log String.

Repopulation

On to the next issue. Clicking the second list box item should refresh the edit fields to display the second record:

```
    TEST_(TestDialog, OnSelChange)
    {
        CListBox aList = m_aDlg.GetDlgItem(IDC_LIST_CUSTOMERS);
        int x = aList.SetCurSel(1);
        CPPUNIT_ASSERT_EQUAL(1, x);
        CPPUNIT_ASSERT_EQUAL("Krazy", m_aDlg.getText(IDC_EDIT_FIRST_NAME));
        CPPUNIT_ASSERT_EQUAL("Kat",   m_aDlg.getText(IDC_EDIT_LAST_NAME));
    }
```

After our journey through low-level GDI MetaFiles, that test looks refreshingly familiar! So does its failure message:

```
    …\project_test.cpp(662) :
  "Krazy"(Krazy) != m_aDlg.getText(IDC_EDIT_FIRST_NAME)(Ignatz)
```

That means `ProjectDlg` needs to handle the `LBN_ONSELCHANGE` notification:

```
        BEGIN_MSG_MAP(ProjectDlg)
…
            COMMAND_ID_HANDLER(IDCANCEL, OnClose)
            COMMAND_HANDLER(IDC_LIST_CUSTOMERS, LBN_SELCHANGE, OnSelChange)
        END_MSG_MAP()

        LRESULT OnSelChange(UINT, WORD, HWND, BOOL &);
        LRESULT OnClose(UINT uCode, int nID, HWND, int);
…
        LRESULT
    ProjectDlg::OnSelChange(UINT, WORD, HWND, BOOL &)
    {
        CListBox aList = GetDlgItem(IDC_LIST_CUSTOMERS);
        int idx = aList.GetCurSel();
        m_aCA.setRecordIndex(idx + 1);
        transferXML(&Field::dialogFromXML);
        return 0;
    }
```

The new `OnSelChange()` method works correctly when we manually test. Clicking on an item in the list box triggers `OnSelChange()`, which refreshes the subservient edit fields. But the automated test, attempting the exact same procedure, fails. (The operation was a failure but the patient got better!) Investigating this issue illustrates how frequent *Temporary Interactive Tests* assist the *User Simulation* Principle.

Loose MS Windows User Simulation

The test calls the WTL wrapper method `.SetCurSel()`. That sends the message `LB_SETCURSEL`, which ostensibly works the same as clicking a list box item to select it. But unlike a real click, the list box's `LB_SETCURSEL` event handler does not bounce any `LBN_SELCHANGE` notification back to the dialog. A click from a real user would notify the dialog that its control changed.

The list box censors the notification to assist programmers using `LB_SETCURSEL` in production code. If the message notified the dialog, that would require excess code to distinguish production code from users selecting list box items. Production code that calls `LB_SETCURSEL` should not accidentally *Simulate User Input*, so this convenience simplifies all that code.

We could defeat this convenience with *Firm User Simulation*. A test fixture could send `WM_NOTIFY` with the control ID and `LBN_SELCHANGE` in the payload. Such a test would constrain this totally obvious code:

```
        BEGIN_MSG_MAP(ProjectDlg)
...
            COMMAND_HANDLER(IDC_LIST_CUSTOMERS, LBN_SELCHANGE, OnSelChange)
        END_MSG_MAP()
```

Instead, we apply the same work-around as production code would use. We simply call `OnSelChange()` directly:

```
    TEST_(TestDialog, OnSelChange)
    {
        CListBox aList = m_aDlg.GetDlgItem(IDC_LIST_CUSTOMERS);
        int x = aList.SetCurSel(1);
        CPPUNIT_ASSERT_EQUAL(1, x);

        BOOL b(FALSE);
        m_aDlg.OnSelChange(0,0,0,b);

        CPPUNIT_ASSERT_EQUAL("Krazy", m_aDlg.getText(IDC_EDIT_FIRST_NAME));
        CPPUNIT_ASSERT_EQUAL("Kat",   m_aDlg.getText(IDC_EDIT_LAST_NAME));
    }
```

The tests pass. Also, as promised on page **Error! Bookmark not defined.**, we started using the new method `setRecordIndex()`. Avid readers may have forgotten about that method. This illustrates the danger of speculative code. Suppose I predict the need for a method, develop it, and then spend time on another task. If that task takes a while, then if the project finds no need for the speculative method, I could forget to remove it. Any future programmer who then tries to use that method, expecting business value has constrained it, might get a nasty surprise.

Before you read that paragraph, did you remember `setRecordIndex()`?

Localization

The word “glyph” has five glyphs and four phonemes. A “phoneme” is the smallest difference in sound that can change a word’s meaning. For example, *f* is softer than *ph*, so *flip* has a meaning different than … you get the idea.

“Ligatures” are links between two glyphs, such as fl, with a link at the top. “Accented” characters, like ā, might be considered one glyph or two. And many languages use “vowel signs” to modifying consonants to introduce vowels, such as the tilde in the Spanish word *niña* (“neenya”), meaning “girl”.

A “script” is a set of glyphs that write a language. A “char set” is a table of integers, one for each glyph in a script. A “code point” is one glyph’s index in that char set. Engineers say “character” when they mean “one data element of a string”, so this book casually uses “character” to mean either 8-bit `char` elements or 16-bit `wchar_t` elements. An “encoding” is a way to pack a char set as a sequence of characters, all with the same bit-count. A “code page” is an identifier to select an encoding. A “glossary” is a list of useful phrases translated into two or more languages. A “collating order” sorts a cultures’ glyphs so readers can find things in lists by name. A “locale” is a culture’s script, char set, encoding, collating order, glossary, icons, colors, sounds, formats, and layouts, all bundled into a seamless GUI experience.

To internationalize, enable the narrowest set of scripts and glossaries that address immediate business needs.

Teams may need to prepare code for glossaries scheduled to become available within a few iterations. Ideally, adding a new locale should require only authoring, not reprogramming. New locales should reside in pluggable modules, so adding them requires no changes to the core source code. The application should be ready for any combination of glossaries and scripts, within business’s short-term goals.

If the business side will only target a certain range of locales, only prepare the code for their kinds of encodings; no more. To target only one range of cultures, such as Western Europe, localize to two glossaries within one script, such as English and Spanish. When other nearby locales, such as Dutch or French, become available, they should plug-and-play. (And remember Swedish has a slightly different collating order!)

If business’s short-term goals specify only languages within one script, such as English and Spanish, code must not prepare for locales with different scripts, such as Manchu or Sylheti. Do not write speculative code that “might” work with other scripts’ encodings, to anticipate a distant future when your leaders request them. Code abilities that stakeholders won’t pay attention to add risk. In our driving metaphor, the project now drives fast in a direction the drivers are not looking.

To target the whole world, before getting all its glossaries, localize to at least 4 scripts, including a right-to-left script and an ideographic one.

Right-to-left scripts require Bi-Directional communication support (BiDi), so embedded left-to-right verbiage “flows” correctly within the right-to-left text. Ideographs overflow naïvely formatted, terse English resource templates (like mine in the last Case Study). To avoid speculation, one should at least localize to enough different kinds of scripts to fully vet the locale, encoding, and display functions’ abilities.

Finally, if the business plan requires only one locale, then you lack the mandate to localize. Hard-code a single locale. Only prepare for the future with cheap and foolproof systems, such as `TEXT()` or `LoadString()`. You aren’t going to need the extra effort and risk of more complex

facilities, like preemptive calls to `WideCharToMultiByte()`. Test-First Programming teaches us the risk of speculative code by lowering many other risks so it sticks out. Bugs hide in code written without immediate tests, reviews, or releases. When new features attempt to use this unproven code, its bugs bite, and lead to those long arduous bug-hunts of the bad old days.

> Do the simplest thing that could possibly work.

Some Agile literature softens that advice to "*Consider* the simplest thing…" That verbiage denies live source code the opportunity to experience the simplest thing, if you can find it. Seeking simplicity in a maze of absurdly complex systems, such as locales, requires capturing that simple thing, when found. Don't "consider" it, DO it!

Write simple code with lots of tests, and keep it easy to refactor and refeaturize. If your application then becomes successful enough to deliver outside your initial target cultures, and if you scrupulously put all strings into the Representation Layer and unified all their duplications, then you will find the task of collecting strings and moving them into pluggable string tables relatively easy.

For our narration, I picked a single target locale with sufficient challenges. Your project, on your platform, will require many more tests than this project can present.

> Escalate any mysterious display bugs into tests that constrain your platform's fonts, code pages, and encodings.

Fonts resist tests. GDI primitives, such as `TextOut()`, cannot report if they drew a □ "Missing Character Glyph". After this little project, we will concoct a way to detect those without visual review.

Localizing to ???????

Sanskrit is an ancient and living language occupying the same roles in Southern Asia as Latin occupies in Southern Europe. We now tackle a fictitious user story "Localize to Sanskrit", and in exchange for some technical challenges, it returns impressive visual results.

Locale Skins

Our first step authors a new locale into our RC file. Copy the `IDD_PROJECT DIALOGEX` and paste it at the bottom of the RC file. Then declare a different locale above it, and put an experimental change into it:

```
LANGUAGE LANG_SANSKRIT, SUBLANG_DEFAULT

IDD_PROJECT DIALOGEX 0, 0, 294, 122
STYLE DS_SETFONT | DS_MODALFRAME | DS_FIXEDSYS | WS_POPUP |
    WS_CAPTION | WS_SYSMENU
CAPTION "Snoopy"
FONT 8, "MS Shell Dlg", 400, 0, 0x1
BEGIN
...
END
```

Resource segments follow a top-level structure of locales, with resource definitions—menus, dialogs, accelerators, etc.—duplicated inside each locale. (This kind of duplication is not as odious as duplicated definitions of behavior; most resources only contain definitions of authorable esthetics and structure. We will eventually observe the need to author new controls twice, and our test rig will help remind us.)

WinXP processes inherit their default locale from Registry settings controlled by the Desktop Control Panel's Regional and Language Options applet. Our tests must not require manual intervention, including twiddling that applet or rebooting. While our Sanskrit skin develops, no bugs must get under our English skin.

So `lstrcmpW()` is an example of a technique, close to the application, that accurately simulates looking at a GUI.

This test suite adjusts the behavior of `TestDialog` (using the Abstract Template Pattern, again), to call `SetThreadLocale()`. That overrides the Control Panel and configures the current thread so any future resource fetches seek the Sanskrit skin first. The only Sanskrit-specific resource is our new `IDD_PROJECT`. Any other fetches shall default back to the English skin.

```
    struct
TestSanskrit:  virtual TestDialog
{

    void
setUp()
    {
    WORD sanskrit(MAKELANGID(LANG_SANSKRIT, SUBLANG_DEFAULT));
    LCID lcid(MAKELCID(sanskrit, SORT_DEFAULT));
    ::SetThreadLocale(lcid);
    TestDialog::setUp();
    }

    void
tearDown()
    {
    TestDialog::tearDown();
    WORD locale(MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT));
    LCID lcid(MAKELCID(locale, SORT_DEFAULT));
    ::SetThreadLocale(lcid);
    }

};
```

The new suite calls back to its base class's `TestDialog::setUp()` and `TestDialog::tearDown()` methods. When they create the dialog member object, its resources select Sanskrit. After the dialog destroys, the suite restores the locale to your desktop's default.

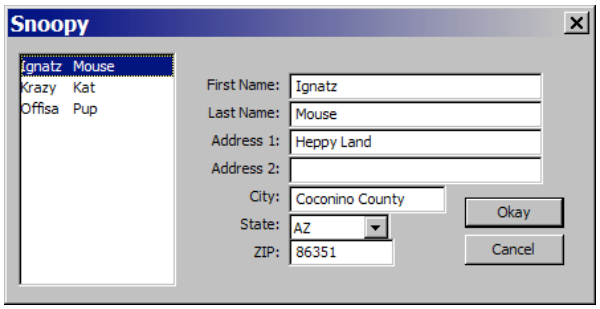
Michael Kaplan, the author of *Internationalization with Visual Basic*, reminds us `SetThreadLocale()` isn't stable enough for production code. While tests may relax these restrictions, industrial-strength localization efforts, on MS Windows, should separate locales into distinct RC and DLL files, one set per target. A test suite should re-use the production code methods that call `LoadLibrary()` to plug these resources in before displaying GUI objects.

This Case Study targets only a few of internationalization's common problems, to bring your platform's best practices as close as a few refactors.

Here's a temporary test using the new suite, and its result:

```
TEST_(TestSanskrit, reveal)
{
```

```
    revealFor("phlip");
}
```



Even a Sanskrit acolyte checking this output could tell that "`Snoopy`" is not Sanskrit. (Trust I really *did* the simplest thing that could possibly work!)

Any result except an easily recognized English name would raise an instant alarm. Changing the new resource incrementally helps us write a *Temporary Visual Inspection* that answers only one question: How to change a locale on the fly, without rebooting? Seeing only one change, "`Snoopy`" for "`Project`" in the window caption, assures us the new resource works, and the derived test suite works. Adding lots of Sanskrit would risk many different bugs, all at the same time. All localization efforts have a high risk of trivial bugs that resist research, and testing.

Babylon

In the beginning, there was ASCII, based on encoding the Latin alphabet, without accent marks, into a 7-bit protocol. Early systems reserved the 8th bit for a parity check.

Then cultures with short phonetic alphabets computerized their own glyphs. Each culture claimed the same "high-ASCII" range of the 8 bits in a byte—the ones with the 8th bit turned on.

User interface software, to enable more than one locale, selects the "meaning" of the high-ASCII characters by selecting a "code page". On some hardware devices, this variable literally selected the hardware page of a jump table to convert codes into glyphs.

Modern GUIs still use code page numbers, typically defined by the "International Standards Organization", or its member committees. The ISO 8859-7 encoding, for example, stores Latin characters in their ASCII locations, and Greek characters in the high-ASCII. Internationalize a resource file to Greek like this:

```
LANGUAGE LANG_GREEK, SUBLANG_NEUTRAL
#pragma code_page(1253)

STRINGTABLE DISCARDABLE
BEGIN
    IDS_WELCOME       "Υποδοχή στην Ελλάδα."
END
```

The quoted Greek words might appear as garbage on your desktop, in a real RC file, or in a compiled application. On WinXP, fix this by opening the Regional and Language Options applet, and switching the combo box labeled "Select a language to match the language version of the non-Unicode programs you want to use" to Greek.

That user interface verbiage uses “non-Unicode” to mean the “default code page”. When a program runs using that resource, the code page “`1253`” triggers the correct interpretation, as (roughly) ISO 8859-7.

MS Windows sometimes supports more than one code page per locale. The two similar pages, 1253 and ISO 8859-7, differ by a couple of glyphs.

Some languages require more than 127 glyphs. To fit these locales within 8-bit hardware, more complex encodings map some glyphs into more than one byte. The bytes without their 8th bit still encode ASCII, but any byte with its 8th bit set is a member of a short sequence of multiple bytes that require some math formula to extract their actual char set index. These “Multiple Byte Character Sets” support locale-specific code pages for cultures from Arabia to Vietnam.

Code page systems resist polyglot GUIs. You cannot put glyphs from different cultures into the same string, if OS functions demand one code page per string. Code page systems resist formatting text together from many cultures. And Win32 doesn’t support all known code pages, making their glyphs impossible.

Sanskrit shares a very popular script called □□□□□□□□ (“Devanāgarī”) with several other Asian languages. (Watch the movie “Seven Years in Tibet” to see a big ancient document, written with beautiful flowing Devanāgarī, explaining why Brad Pitt is not allowed in Tibet.)

Devanāgarī’s code page could have been 57002, based on the standard “Indian Script Code for Information Interchange”. MS Windows does not support this locale-specific code page. Accessing Devanāgarī and writing Sanskrit (or most other modern Indian languages) requires the Mother of All Char Sets.

Unicode

ISO 10646, and the “Unicode Consortium”, maintain the complete char set of all humanity’s glyphs. To reduce the total count, Unicode supplies many shortcuts. For example, many fonts place glyph clusters, such as accented characters, into one glyph. Unicode usually defines each glyph component separately, and relies on software to merge glyphs into one letter. That rule helps Unicode not fill up with all permutations of combinations of ligating accented modified characters.

Many letters, such as ñ, have more than one Unicode representation. Such a glyph could be a single code point (`L"\xF1"`), grandfathered in from a well-established char set, or could be a composition of two glyphs (`L"n\x303"`). The C languages introduce 16-bit string literals with an `L`.

Text handling functions must not assume each data character is one glyph, or compare strings using naïve character comparisons. Functions that process Unicode support commands to merge all compositions, or expand all compositions.

The C languages support a 16-bit character type, `wchar_t`, and a matching `wcs*()` function for every `str*()` function. The `strcmp()` function, to compare 8-bit strings, has a matching `wcscmp()` function to compare 16-bit strings. These functions return `0` when their string arguments match.

(Another point of complexity; I will persist in referring to `char` as 8 bit and `wchar_t` as 16-bit, despite the letters of the C Standard law say they may store more bits. These rules permit the C languages to fully exploit various hardware architectures.)

Irritatingly, documentation for `wcscmp()` often claims it can compare "Unicode" strings. This Characterization Test demonstrates how that claim misleads:

```
TEST_(TestCase, Hoijarvi)
{
    std::string str("Höijärvi");
    WCHAR composed[20] = {0};

    MultiByteToWideChar(
                CP_ACP,
                MB_COMPOSITE,
                str.c_str(),
                -1,
                composed,
                sizeof composed
                );
    CPPUNIT_ASSERT(0 != wcscmp(L"Höijärvi", composed));
    CPPUNIT_ASSERT(0 == wcscmp(L"Ho\x308ija\x308rvi", composed));
    CPPUNIT_ASSERT(0 == lstrcmpW(L"Höijärvi", composed));

    CPPUNIT_ASSERT_EQUAL
        (
        CSTR_EQUAL,
        CompareStringW
            (
            LOCALE_USER_DEFAULT,
            NORM_IGNORECASE,
            L"höijärvi", -1,
            composed, -1
            )
        );
}
```

The test starts with an 8-bit string, `"Höijärvi"`, expressed in my editor's code page, ISO 8859-1, also known as Latin 1. Then `MultiByteToWideChar()` converts it into a Unicode string with all glyphs decomposed into their constituents.

The first assertion reveals that `wcscmp()` compares raw characters, and thinks `"ö"` differs from `"o\x308"`, where `\x308` is the COMBINING DIAERESIS code point.

The second assertion proves the exact bits inside `composed` contain primitive `o` and `a` glyphs followed by combining diæreses.

This assertion…

```
    CPPUNIT_ASSERT(0 == lstrcmpW(L"Höijärvi", composed));
```

…reveals the MS Windows function `lstrcmpW()` correctly matches glyphs, not their constituent characters.

The long assertion with `CompareStringW()` demonstrates how to augment `lstrcmpW()`'s internal behavior with more complex arguments.

Unicode Transformation Format

`wchar_t` cannot hold all glyphs equally, each at their raw Unicode index. Despite Unicode's careful paucity, human creativity has spawned more than 65,535 code points. Whatever the size of your characters, you must store Unicode using its own kind of Multiple Byte Character Set.

UTF converts raw Unicode to encodings within characters of fixed bit widths. UTF-7, UTF-8, UTF-16, UTF-32, all may store any glyph in Unicode, including those above the 0xFFFF mark.

MS Windows, roughly speaking, represents UTF-8 as a code page among many. However, roughly speaking again, when an application compiles with the `_UNICODE` flag turned on, and executes on a version of Windows derived from WinNT, it obeys UTF-16 as a code page, regardless of locale.

Because a `_UNICODE`-enabled application can efficiently use UTF-16 to store a glyph from any culture, such applications needn't link their locales to specific code pages. They can manipulate strings containing any glyph. In this mode, all glyphs are created equal.

`_UNICODE`

Resource files that use UTF-8 configure their 8-bit code pages with `#pragma code_page(#)`. When a resource file saves in UTF-16 format, the resource compiler, `rc.exe`, interprets RC files stored in UTF-16 text format as a global code page covering all locales. Before tossing □□□□□□□□ into our resource files, our program needs a "refactor" to use this global code page.

Switch Project → Project Properties → General → Character Set to "Use Unicode Character Set". That turns on the compilation conditions `UNICODE` and `_UNICODE`. Recompile, and get a zillion trivial syntax errors.

You might want to integrate before those changes, to create a roll-back point if something goes wrong.

When `CString` sees the new `_UNICODE` flag, the `XCHAR` inside it changes from an 8-bit `CHAR` to a 16-bit `WCHAR`. That breaks typesafety with all characters and strings that use an 8-bit `char`. Fix many of these errors by adding the infamous `TEXT()` macro, and by using the `W`ide version of our test macros. Any string literal that interacts with `CString`s needs this treatment:

```
TEST_(TestDialog, changeName)
{
  m_aDlg.SetDlgItemText(IDC_EDIT_FIRST_NAME, TEXT("Krazy"));
  m_aDlg.SetDlgItemText(IDC_EDIT_LAST_NAME,  TEXT("Kat"));
  CustomerAddress &aCA = m_aDlg.getCustomerAddress();
  m_aDlg.saveXML();

  CPPUNIT_ASSERT_EQUAL_W( TEXT("Krazy"),
                  aCA.get(TEXT("first_name")) );

  CPPUNIT_ASSERT_EQUAL_W( TEXT("Kat"),
                  aCA.get(TEXT("last_name"))  );
}
```

This project used no unnecessary typecasts. A stray `(LPCTSTR)` typecast in the wrong place would have spelled disaster, because the `T` converts to a `W` under `_UNICODE`. `(LPCWSTR)"Krazy"` does not convert "`Krazy`" to 16-bit characters; it only forces the compiler to disregard the "`Krazy`" characters' true type. C++ permits easy typecasts that can lead to undefined behavior.

Using types safely, without typecasts, permits the compiler's syntax errors to navigate to each simple change; typically lines with string literals like these, that need `TEXT()` macro calls:

```
aDCmeta.Create(aDC, TEXT("sample.emf"), &rc, TEXT("test"));
```

Lines that use `_bstr_t` won't need many changes, because its methods overload for both wide and narrow strings. And some few `CString`s should remain narrow. They could convert to `CStringA`, but we will use `std::string` for no reason:

```
    std::string
readFile(char const * fileName)
{
    std::string contents;
    std::ifstream in(fileName);
    char ch;
    while (in.get(ch))  contents += ch;
    return contents;
}
```

And the assertions need a new, type-safe stream insertion operator:

```
   inline std::wostream &
operator<<(std::wostream &o, CStringW const &str)
{
    CString copy(str);
    if (str.GetLength())  o << copy.GetBuffer(0);
    return o;
}
```

When you make all these changes, if your project gets stuck, switch `_UNICODE` off, and run all the tests. If they fail, erase your source and check out the latest version from your version controller. Then read *Developing International Software* by Dr. International, and apply the 9 step procedure entitled "Migration to Unicode". It provides many more opportunities for all your tests to pass. But the book does not specify "run all your tests now", after each step. Probably just an oversight by the author.

> Clean code accepts major, invasive changes gracefully & incrementally.

When legacy projects suffer during internationalization, they reveal their own hidden cruft. Don't blame the messenger.

Spiderman

Our current source code started, two Case Studies ago, with a simple test rig and a plain dialog. Then our tests prepared for localization by deriving a new suite from `TestDialog` and switching its locale to Sanskrit. Then our production code prepared for localization by turning on the Unicode system. We are still not ready to localize. We have not yet determined if we could recognize a garbage-in-garbage-out situation if we saw one. Sanskrit is Greek to many developers. Test-First Programming relies on developers who frequently question their own tests' validity.

Our next step changes our window's caption from "Snoopy" into something written in simple phonetic Devanāgarī. Engineers learn their project's domain as they write features, and they learn enough of their target languages, as they internationalize, to support brief visual inspections. Phonetic languages make converting words to sounds easy, so start your learning here.

Open your Web browser, and hit your target locale's BBC web site. Devanāgarī appears on the Hindi skin of BBC News. Seek cheerful news. On some days, only advertisements or movie reviews qualify. Copy sample words out, such as □□□□□□□□□□□, and paste them into

Notepad. (Both the BBC News Websites, and the lowly MS Windows Notepad.exe program, provide exemplary internationalization, so learn to leverage both.) Learn to read the characters you copied, by translating them, or decoding their phonemes. Write down the translation for later comparison.

Save the sample words as `sample.txt`, with the "Unicode big endian" encoding. Apply a program that converts binary files to hexadecimal to `sample.txt`:

```
  Addr     0 1  2 3  4 5  6 7  8 9  A B  C D  E F 0 2 4 6 8 A C E
--------  ---- ---- ---- ---- ---- ---- ---- ---- ----------------
00000000  feff 0938 094d 092a 093e 0907 0921 0930 ~..8.M.*.>...!.0
00000010  092e 0948 0928                         ...H.(
```

The "Byte Order Mark", `FeFF`, is one of a few "Magic Numbers" that trigger Unicode parsing for text files. Convert all the codes after that BOM into a wide string literal:

```
L"\x0938\x094D\x092A\x093E\x0907\x0921\x0930\x092E\x0948\x0928"
```

We did all that to avoid pasting non-ASCII directly into program source. The Visual Studio IDE accepts literal 16-bit Unicode characters, if you save their files as Unicode. This Case Study illustrates doing these things the hard way, because sometimes programmers must.

Now write a test that demands our window's caption contains the text sample:

```
TEST_(TestSanskrit, reveal)
{

    CStringW caption;
    m_aDlg.GetWindowText(caption);

    CStringW expect =
L"\x0938\x094D\x092A\x093E\x0907\x0921\x0930\x092E\x0948\x0928";

    CPPUNIT_ASSERT_EQUAL(0, lstrcmpW(expect, caption));
    revealFor("phlip");
}
```

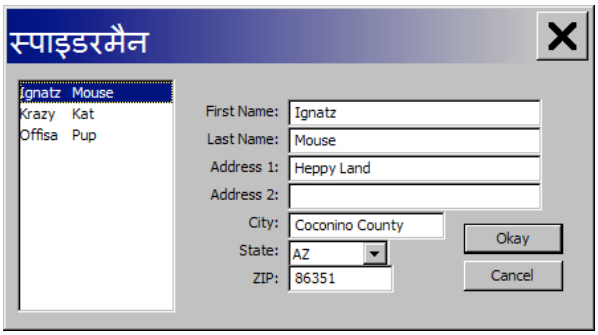
Note that big string does not use the `TEXT()` macro. The width of its characters is not optional.

There are two ways to pass that test. Either paste raw Devanāgarī directly into your resource file, or paste the escaped codes. Linguists naturally prefer the first way, and programmers must also learn the second, so I commented the first way out:

```
LANGUAGE LANG_SANSKRIT, SUBLANG_DEFAULT

IDD_PROJECT DIALOGEX 0, 0, 294, 122
STYLE DS_SETFONT | DS_MODALFRAME | DS_FIXEDSYS | WS_POPUP | WS_CAPTION |
    WS_SYSMENU
//CAPTION "□□□□□□□□□□□□"

CAPTION
L"\x0938\x094D\x092A\x093E\x0907\x0921\x0930\x092E\x0948\x0928"

FONT 16, "MS Shell Dlg", 400, 0, 0x1

BEGIN
```

```
...
    END
```

The next step switches the resource to the correct code page. From the file `Project.rc`, select File → Advanced Save Options → Unicode – Codepage 1200. The file will declare itself modified. Save it and run all the tests.



The font `"MS Shell Dlg"` collates glyphs from many similar fonts, to present a globalized super-font. Without it, our project would need a custom font.

I changed my Desktop's Display Properties to make the title bar large, and not **bold**, so we can see the details. Phonetically, we have:

- □□□ **spa**
- □ *raises the* a *toward* **i**
- □ **ee**
- □ **d**
- □ **ar**
- □ **m**
- □ *adds* **ai**
- □ **n**

Note the ligature between □ *sa* and □ *pa*, forming □□□ *spa*. That ligature suppressed the default *a* sound in □ *sa*. The little □ things are accent marks decomposed from their target glyphs. The dotted circle represents their missing target.

Assemble those phonemes to reveal the name of a mythical creature who is half human, half arachnid.

Glossaries

This has been a *Temporary Visual Inspection* to ensure we can localize accurately. We would be in trouble if the above experiment had revealed "The Hulk".

We would also be in trouble if such experiments escaped our lab. Page 51 reminds us not to integrate aggressive experiments. Erase this change and start again with a real glossary, in modern Sanskrit:

- □□□□ □□□ *upakramaH* project
- □□□ *naama* name
- □□□□ □□ *pulanaama* surname
- □□□□ □, □ *viithi* road 1, 2
- □□□□ □□□ *graamam* village
- □□□□ *deshe* land
- □□□□ □□ *sa.nhitaa* code
- □□□□ *traa* save
- □□□□ □ *viraama* stop

Enter these into your resource files using one of two systems. Install a driver, or "Input Method Editor", that maps your keyboard onto Devanāgarī, and type the symbols into your editor. Alternately, look each component of each letter up in a Unicode reference, and enter their numbers as string literal escape codes. Most Unicode points form acceptable UTF-16 codes; only the ones near or above 65,535 (`0xFFFF`) need transformations.

Use the points to enter wide strings like these:

```
L"\x0935\x093F\x0930\x093E\x092E"
```

That spells *viraama*, and it illustrates a curious point about vowel signs. The first two codes are:

- `\x0935` □
- `\x093F` □

The symbol □ *i* modifies the glyph to its right, but its `\x093F` goes *after* the `\x0935` □ *v*. Unicode puts all base glyphs to the left and their modifiers on their right.

(Ironically, Devanāgarī pronounces the resulting *iv* as *vi*, so Unicode happens to match how readers pronounce these glyphs!)

As a programmer, I need a resource file that makes all these details explicit, so we will author the new strings into our `LANG_SANSKRIT` locale using only their hexadecimal representations, not their glyphs. Your team may disagree, and put glyphs directly in resource files. Whatever format you use, you should boost your confidence by writing a quicky test function to convert between formats. My `TEST_(TestCase, Hoijarvi)` illustrated MS Windows's conversion functions; they should go into test fixtures for rapid review. (And

thanks to Kari Höijärvi, on the TFUI mailing list, for pointing out the effect his name has on internationalization efforts!)

Another reason to leave Devanāgarī, specifically, out of your source code is that the language requires some wild descenders, such as ??. Either all these letters appear very small, or you must set your font to very large.

Spot Checks

To spot-check this change, we will upgrade our experimental test into a real test that checks the title bar, and the Save button. Tests like these are insufficient to constrain this entire feature; hence this Case Study's name.

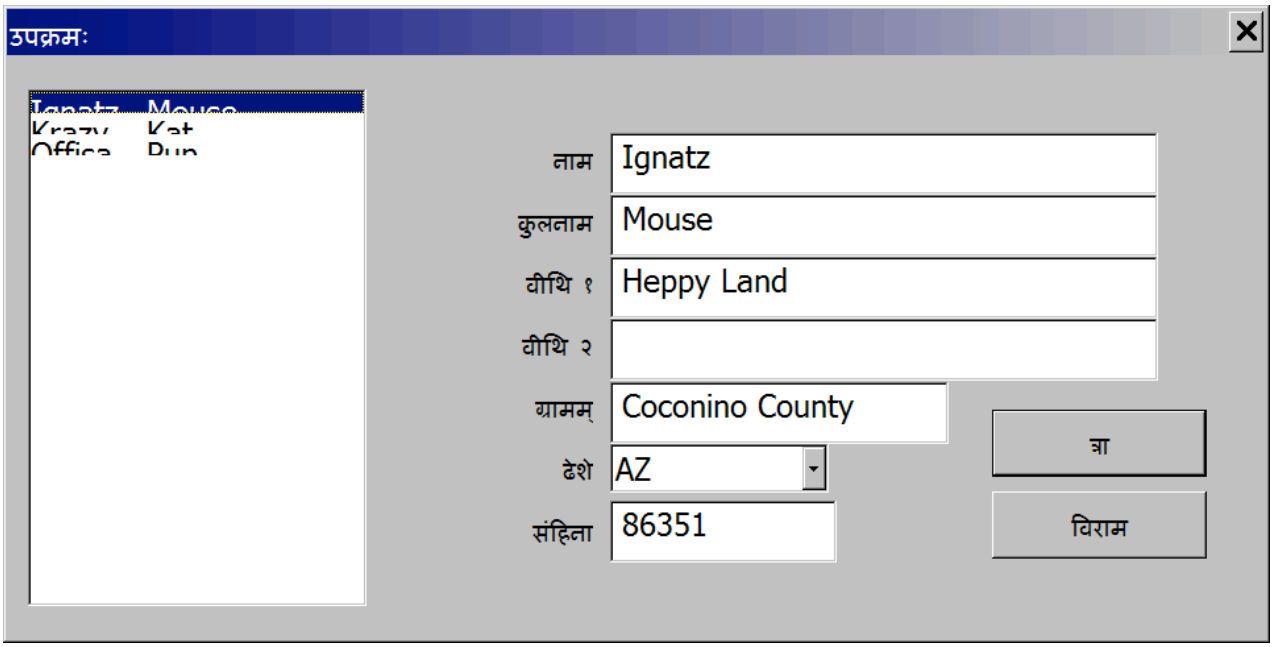
```
TEST_(TestSanskrit, caption)
{

    CStringW caption;
    m_aDlg.GetWindowText(caption);

    CStringW expect =
            L"\x0909\x092A\x0915\x094d\x0930\x092e\x0903";
                // upkraamaH—project

    CPPUNIT_ASSERT_EQUAL(0, lstrcmpW(expect, caption));

    expect = L"\x0924\x094d\x0930\x093e";  //  traa—save
    m_aDlg.GetDlgItemText(IDOK, caption);
    CPPUNIT_ASSERT_EQUAL(0, lstrcmpW(expect, caption));

    revealFor("phlip");

}
```

If your team needs glyphs in resource files, you must use your conversion program to read resource strings and reveal their hex codes to write such tests. I pasted strings into Notepad, saved them as "Unicode – big endian", and ran `dump.exe` to output their hexadecimal values. Then I typed all their raw codes in:

```
LANGUAGE LANG_SANSKRIT, SUBLANG_DEFAULT

IDD_PROJECT DIALOGEX 0, 0, 294, 122
STYLE DS_SETFONT | DS_MODALFRAME | DS_FIXEDSYS ...
CAPTION L"\x0909\x092A\x0915\x094D\x0930\x092E\x0903"
FONT 18, "MS Shell Dlg", 400, 0, 0x1

BEGIN
    LISTBOX      IDC_LIST_CUSTOMERS,5,7,79,108,LBS_SORT |
                       LBS_NOINTEGRALHEIGHT |
                       WS_VSCROLL | WS_TABSTOP |
                       LBS_OWNERDRAWFIXED | LBS_HASSTRINGS

    RTEXT      L"\x0928\x093E\x092E",IDC_STATIC, ...
    EDITTEXT   IDC_EDIT_FIRST_NAME, ...
    RTEXT      L"\x0915\x0941\x0932\x0928\x093E\x092E", ...
    EDITTEXT   IDC_EDIT_LAST_NAME, ...
```

```
    RTEXT L"\x0935\x0940\x0925\x093F\x0967",IDC_STATIC, ...
    EDITTEXT    IDC_EDIT_ADDRESS_1, ...
    RTEXT       L"\x0935\x0940\x0925\x093F\x0968", ...
    EDITTEXT    IDC_EDIT_ADDRESS_2, ...
    RTEXT L"\x0917\x094D\x0930\x093E\x092E\x092E\x094D", ...
    EDITTEXT    IDC_EDIT_CITY, ...
    RTEXT       L"\x0922\x0947\x0936\x0947", ...
    COMBOBOX    IDC_COMBO_STATE, ...
    RTEXT   L"\x0938\x0902\x0939\x093F\x0928\x093E", ...
    EDITTEXT        IDC_EDIT_ZIP, ...
    DEFPUSHBUTTON   L"\x0924\x094D\x0930\x093E",IDOK, ...
    PUSHBUTTON L"\x0935\x093F\x0930\x093E\x092E",IDCANCEL, ...
END
```

The new skin reveals a display bug in the list box:



I switched the font size to `18` to show details. The bit `LBS_OWNERDRAWFIXED` assumes the font size `8`. Fix this inside `OnInitDialog()`. This (authored) code selects the current font, extracts its metrics, adds a little fudge factor, and configures the list box to use a height based on the current skin:

```
    LRESULT
ProjectDlg::OnInitDialog(UINT, WPARAM, LPARAM, BOOL &)
{
    CListBox    aList     = GetDlgItem(IDC_LIST_CUSTOMERS);
   ...

    if (LANG_SANSKRIT == PRIMARYLANGID(::GetThreadLocale()))
        {
        if (!m_fontSanskrit.m_hFont)
            m_fontSanskrit.
                CreatePointFont(180, TEXT("Sanskrit 2003"));
```

```
            CClientDC aDC(this->m_hWnd);

            CFontHandle oldFont = aDC.SelectFont(m_fontSanskrit);
            TEXTMETRIC metrics;
            aDC.GetTextMetrics(&metrics);
            aDC.SelectFont(oldFont);
            aList.SetItemHeight(0, metrics.tmHeight);

            CWindow first = GetWindow(GW_CHILD);
            CWindow next = first;

            do {
                next.SetFont(m_fontSanskrit);
                next = next.GetWindow(GW_HWNDNEXT);
                } while (next.m_hWnd);
            }
        return 0;
    }
```

For one last flourish, I slipped in a beautiful font called "Sanskrit 2003", by the studious Omkarananda Ashram Himalayas. I could have added it to the dialog resource; instead I loop through each dialog item, and push the new font into it.

The final product (enlarged to show texture):

Our small tests are sufficient for TFP and refactoring. However, even a test that compared every string would not be sufficient for an entire localization project. I copied the same hex codes into both the test and the resource file. If I encoded a mistake, such as "□ □", to me it would look like a harmless "`\x093F\x0935`". I would then copy it into both places, and the test would reinforce the mistake, not catch it.

Missing Character Glyphs

Your linguists cannot test-first every glyph, and if you don't understand the glyphs then you *shouldn't* test-first them. To constrain these risks, acceptance tests can scan all the text in your application and perform general sanity checks. Languages exist to be parsed. A high-content system like a Web site should use a grammar checker and spelling checker for each language. Engineers should support linguists, and refactoring, by adding tests that perform simple checks on content.

In theory, checking that no window displays a Missing Character Glyph, □, should be simple. The low-level methods that might produce this glyph, such as `TextOut()`, know they drew it. But they won't tell the programmer. Put another way, the GDI layer has the ability to Set a string containing potential text, but misses the ability to Get the information that the string was wrong. GDI painted this information on the screen, then threw it away.

We could address this by opening our font files, decoding them, finding all their glyphs (and their ligatures, modifiers, etc.), summating all this information, and comparing it to our text resources. That effort would duplicate the activities of the font manager inside GDI.

In a pinch, one can borrow methods from unused libraries to boost testage. Libraries are easier to add to test code than production code. The library "Uniscribe" supports editors that enable users to write partial and then complete ligating glyphs, no matter what their intermediate shapes. The library manipulates every glyph in a font, so tests on such glyphs might borrow its support utilities.

This book must neglect Uniscribe's core features, following the business assumption that your users already have "Input Method Editors" for their home locales. If you need a Uniscribe-enabled GUI layer, start with *Temporary Interactive Tests* that activate localized keyboard layouts.

This test shows Uniscribe's `ScribeGetCMap()` using our test dialog's device context and font to pass a healthy string, and fail an unhealthy one:

```
#include "Usp10.h"
   //  tell linker to get Uniscribe library
#pragma comment(lib, "Usp10.lib")

    bool
codePointsAreHealthy(CClientDC &aDC, WCHAR * codePoints)
{
    static SCRIPT_CACHE cache = NULL;
    WORD outGlyphs[100] = {0};
    HRESULT hr;

    hr = ScriptGetCMap
            (
            aDC,       // In   Optional device context
            &cache,    // InOut Address of Cache handle
            codePoints,
            wcslen(codePoints),
            0,         // In   Flags such as SGCM_RTL
            outGlyphs  // Out  Array of glyphs
            );

    return S_OK == hr;
}

TEST_(TestSanskrit, ScriptGetCMap)
{

    CListBox aList(m_aDlg.GetDlgItem(IDC_LIST_CUSTOMERS));
    CClientDC aDC(aList);
    CFontHandle font = aList.GetFont();
    aDC.SelectFont(font);

    WCHAR broke[] = L"\x0900";          //  a dead code point
    CPPUNIT_ASSERT(!codePointsAreHealthy(aDC, broke));

    WCHAR vi[] = L"\x0935\x093F";       //  a healthy ligature
    CPPUNIT_ASSERT(codePointsAreHealthy(aDC, vi));

}
```

(Note that Uniscribe only supports Devanāgarī for MS Windows >= 2000, MS Office >= 2000 or Internet Explorer >= 5.0.)

Now that we have the technique, we need a test that iterates through all controls, extracts each one's string, and checks if it contains a dead spot. Put a `\x0900` or similar dead spot into your resource files, in a label, and see if this catches it.

Because this test cycles through every control, it's a good place to add more queries. I slipped in a simple one, `IsTextUnicode()`, as an example:

```
TEST_(TestSanskrit, _checkAllLabels)
{
    CListBox aList(m_aDlg.GetDlgItem(IDC_LIST_CUSTOMERS));
    CClientDC aDC(aList);
    CFontHandle font = aList.GetFont();
    aDC.SelectFont(font);

    CWindow first = m_aDlg.GetWindow(GW_CHILD);
    CWindow next = first;

    do {
        CString text;
        next.GetWindowText(text);

        if (text.GetLength() > 2)
            {
            INT result = IS_TEXT_UNICODE_UNICODE_MASK;
            CString::XCHAR * p = text.GetBuffer(0);
            int len = text.GetLength() * sizeof *p;
            CPPUNIT_ASSERT(IsTextUnicode(p, len, &result));

            CPPUNIT_ASSERT
                (
                codePointsAreHealthy(aDC, p));
            }

        next = next.GetWindow(GW_HWNDNEXT);
        } while (next.m_hWnd);
}
```

That works great—for Sanskrit. What about all the other locales?

Abstract Skin Tests

A GUI with more than one skin needs tests that cover every skin, not just the one currently under development. Refactors and new features in one skin should not break others. Per the practice *Version with Skins* (from page 42), concrete tests for each skin will inherit and specializes a common Abstract Test.

Our `TEST_()` macro needs a tweak to support Abstract Tests. First we switch our latest test to constrain English, because the base class for `TestSanskrit` is `TestDialog`. This refactor moves the case we will abstract up the inheritance graph:

```
TEST_(TestDialog, _checkAllLabels)
{
…
}
```

Now write a new macro that reuses a test case, such as `_checkAllLabels`, into any derived suite, using some good old-fashioned “Diamond Inheritance”:

```
#define TEST_(suite, target)             \
    struct suite##target:  virtual suite \
    { void runCase(); }                  \
    a##suite##target;                    \
    void suite##target::runCase()

#define RETEST_(base, suite, target)    \
    struct base##suite##target:         \
```

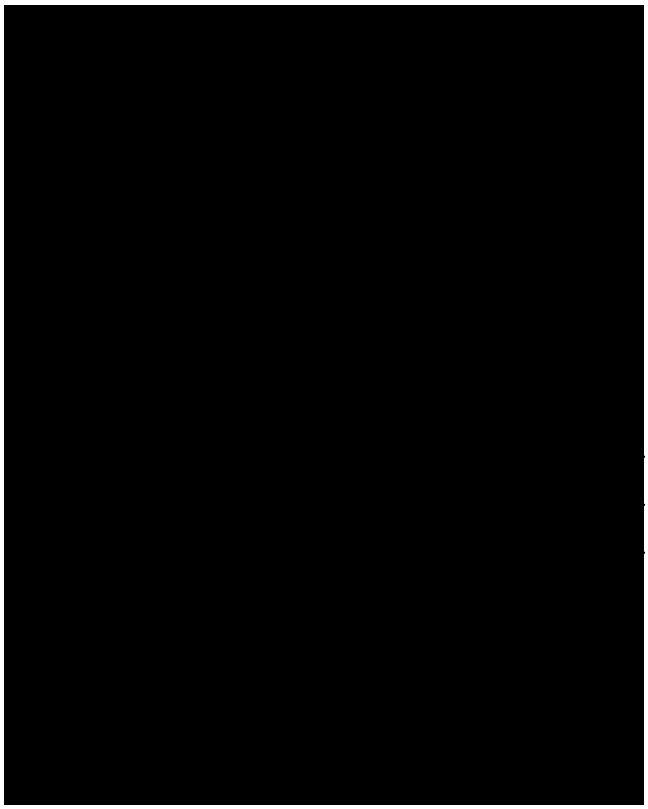
```
        virtual suite,                  \
        virtual base##target  {         \
      void setUp() { suite::setUp(); }   \
      void runCase() { base##target::runCase(); }  \
      void tearDown() { suite::tearDown(); }  \
     } a##base##suite##target;
```

…

Then express that macro with three parameters: The base class, the derived class whose `setUp()` and `tearDown()` we need, and one base class case. The macro reuses that case with the derived class:

```
RETEST_(TestDialog, TestSanskrit, _checkAllLabels)
```

That change required `TestSanskrit` to inherit `TestDialog virtually`, to ensure that `suite::setUp()` sets up the same `m_aDlg` member object as `base##target::runCase()` tests.



Without `virtual` inheritance, C++'s multiple inheritance system makes that chart into a huge V, disconnected at the top. The `TestDialogTestSanskrit_checkAllLabels` object would contain two different `TestDialog` sub-objects, and these would disagree which instance of their member variable `m_aDlg` to test, and which to localize to Sanskrit.

Future extensions could create a `template` that abstracts `setUp()` and `tearDown()` across a list of locales. When the time comes to conquer—oops I mean "support"—the entire world, we should build more elaborate Abstract Tests, then declare stacks of them, one per target locale:

```
RETEST_(TestDialog, TestLocale< LANG_AFRIKAANS >,_checkAllLabels)
RETEST_(TestDialog, TestLocale< LANG_ALBANIAN  >,_checkAllLabels)
RETEST_(TestDialog, TestLocale< LANG_ARABIC    >,_checkAllLabels)
RETEST_(TestDialog, TestLocale< LANG_ARMENIAN  >,_checkAllLabels)
RETEST_(TestDialog, TestLocale< LANG_ASSAMESE  >,_checkAllLabels)
RETEST_(TestDialog, TestLocale< LANG_AZERI     >,_checkAllLabels)
RETEST_(TestDialog, TestLocale< LANG_BASQUE    >,_checkAllLabels)
    …
```

Their test cases should sweep each window and control, for each locale, to check things like overflowing fields, missing hotkeys, etc. Only perform such research as your team appears to need it. (And notice I localized to Sanskrit without adding hotkeys to each label. Only a linguist proficient in a culture's keyboarding practices can assist that usability goal.)

This Case Study pushed the limits of the *Query Visual Appearance* Principle. Nobody should spend all their days researching dark dusty corners of GDI. No trickery in the graphics drivers will rescue usability assessments from repeated painstaking manual review.

This Case Study will add one more feature before making manual review very easy. Simultaneously automating the review of locales and animations separates the acolytes from the □□□□□□ .

Progress Bars

During a long process, users need more than assurance the program did not hang. They need to know how long a process will take, so they can schedule a break, lunch, or a vacation. And they need the option to cancel progress without deleting your program from their task list.

Progress bars' lowly status, as event-driven supplements to procedural features (often added late, after those features work), exposes them to some common AntiPatterns. Don't:

- raise a separate dialog just to display the progress bar
- put the logical procedure into a separate thread
- write a loop statement in the GUI thread to spin until the procedure advances
- guess or fudge the number of ticks to progress—count them
- convert the mouse pointer to an hourglass if clicking still works
- block the event queue, disabling clicking and the `Paint()` event
- display a cancel button that does nothing.

Most of those admonitions have common exceptions. Some procedures by nature are not interruptible. Some, such as Web browsers, communicate with distant unreliable servers, and cannot predict the number of ticks a progress bar will consume.

Our dialog can display customer names and addresses, in two locales, but it does not yet do anything requiring significant time. To require a progress bar, we will add a pushbutton called "Slow Operation". Pushing it will traverse the list of customers, and send each one into a function that does nothing but takes a long time. Then we will write a *Temporary Interactive Test* that populates the list box with many characters, *Simulates a User* starting the "Slow Operation", and observes the progress bar in action.

First, author a pushbutton and a progress bar onto the dialog:

```
    #define IDC_LIST_CUSTOMERS              1008
    #define IDC_SLOW_OPERATION              1011
    #define IDC_PROGRESS                    1012

…
    IDD_PROJECT DIALOGEX 0, 0, 294, 141

…
    BEGIN

…
        DEFPUSHBUTTON   "Okay",IDOK,230,74,50,14
        PUSHBUTTON      "Cancel",IDCANCEL,230,91,50,14
        PUSHBUTTON      "Slow &Operation",IDC_SLOW_OPERATION,
                                              6,120,84,14
        CONTROL         "",IDC_PROGRESS,"msctls_progress32",
                                    WS_BORDER,96,122,183,8
    END
```

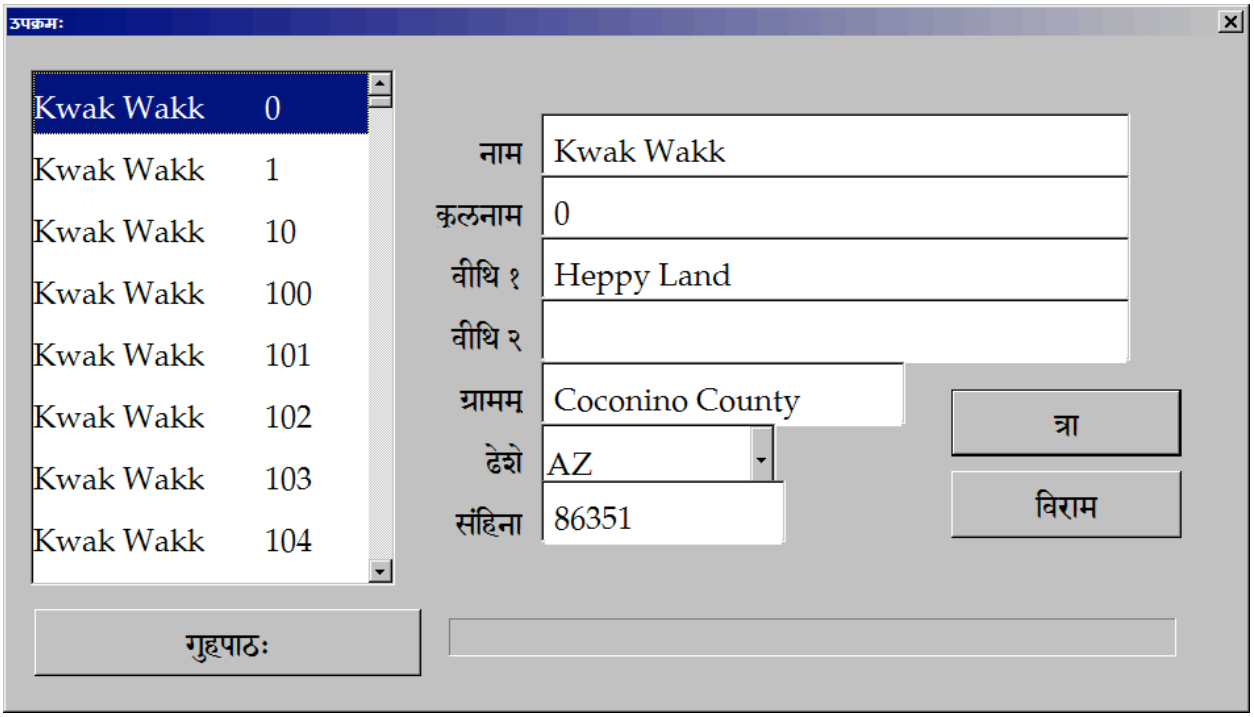
Irritatingly, merely painting an `"msctls_progress32"` control on a dialog is cause for WTL to fail an assertion inside its code. Don't believe any books that say some features can merely be authored. I ran all the tests after painting each new control, so I know which one did it.

The fix is to get inside `main()` and add **`InitCommonControls()`**.

Here's a test case to fill the list box with monotonically numbered customers, and display the dialog:

```
    void
stuffLongUserList(ProjectDlg &aDlg, int max = 1000)
{
    CString xml = "<users>";

    for (int x(0);  x < max;  ++x)
        {
        CString user;
        user.Format
            (
            "<user>"
            "<first_name>Kwak Wakk</first_name>"
            "<last_name>%i</last_name>"
            "<address_1>Heppy Land</address_1>"
            "<address_2></address_2>"
            "<city>Coconino County</city>"
            "<state "
              "options='AK,AL,AR,AZ,'"
            ">AZ</state>"
            "<zip>86351</zip>"
            "</user>",
            x
            );
        xml += user;
        }

    xml += "</users>";
    IXMLDOMDocument2Ptr pXML = aDlg.getCustomerAddress().getXML();
    VARIANT_BOOL parsedOkay = pXML->loadXML(_bstr_t(xml));
    BOOL b(FALSE);
    CListBox aList = aDlg.GetDlgItem(IDC_LIST_CUSTOMERS);
    aList.ResetContent();
    aDlg.OnInitDialog(0,0,0,b);
}
```

```
TEST_(TestDialog, SlowProcess)
{
    stuffLongUserList(m_aDlg);
    revealFor("phlip");
}

RETEST_(TestDialog, TestSanskrit, SlowProcess)
```



The combination of `RETEST_()` and `revealFor()` pops our dialog up twice. If this became an issue, an `if` statement could select only one skin for `revealFor()`.

Revealing all skins reminds us to add the Slow Operation button, and progress bar, to each of them. A larger project would need a more formal review system here.

This code links our `ProjectDlg` to the Slow Operation button:

```
    class
ProjectDlg:
    public CDialogImpl<ProjectDlg>
{
  public:
    enum { IDD = IDD_PROJECT };

    BEGIN_MSG_MAP(ProjectDlg)
...
        COMMAND_HANDLER(IDC_LIST_CUSTOMERS, LBN_SELCHANGE, OnSelChange)
      COMMAND_ID_HANDLER(IDC_SLOW_OPERATION, OnSlowOperation)
    END_MSG_MAP()

    LRESULT OnSlowOperation(WORD, WORD, HWND, BOOL &);
...
};
```

Taking care of that busywork lets us try a naïve implementation of a progress bar:

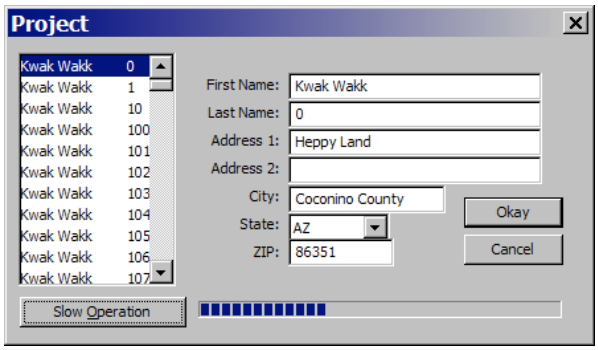
```
    LRESULT
ProjectDlg::OnSlowOperation(WORD, WORD, HWND, BOOL &)
{
    CListBox aList = GetDlgItem(IDC_LIST_CUSTOMERS);
    int count(aList.GetCount());

    CProgressBarCtrl aBar = GetDlgItem(IDC_PROGRESS);
    aBar.SetRange(0, count);
    aBar.SetPos(0);

    for (int x(0);  x < count;  ++x)
        {
        Sleep(25);
        aBar.SetPos(x);
        }

    return 0;
}
```

The function `Sleep(25)` simulates some Logic Layer function that slowly processes one record.

That design is not good enough yet, but it could be worse. The design could call `aBar.SetPos(x)` from somewhere deep inside the function that `Sleep(25)` represents. That would couple the Logic Layer to a GUI Layer identifier, `aBar`. This design decouples a little, but tests will soon force it to decouple more.

This *Temporary Interactive Test* shows the bar trundling slowly across the screen:

```
TEST_(TestDialog, SlowOperation)
{
    stuffLongUserList(m_aDlg);
    CButton slowButton = m_aDlg.GetDlgItem(IDC_SLOW_OPERATION);
    slowButton.PostMessage(BM_CLICK);
    revealFor("phlip");
}
```



`OnSlowOperation()`'s design prevents interaction. After starting the slow process, the program's event queue ought to function. Clicking the "Slow Operation" button again must cancel the operation. (And a polished GUI would change that button's label to "Cancel Operation", to prompt the user.)

If that loop called a method like "`DoEvents()`" (containing `PeekMessage()` and `DispatchMessage()`), the event queue would work, but the design would not get better. It would couple the loop driver to the looped action. A better design supports both users and tests.

`OnSlowOperation()`'s loop statement cannot easily insert test code between each tick of the loop. Tests force narrow dependencies between modules; the inability to test is a sign of a poor design. Our loop should not use `DoEvents()` to pump the message queue; the loop should not duplicate the behavior of our application's real event queue. MS Windows enables subtle bugs when two event queues pumps disagree on each message's destiny.

To pull this problem inside out, use a window timer. We will call `SetTimer()` to tell the OS how often to send a `WM_TIMER` message, then handle it. This lets our message queue pump our loop. That's better than our loop pumping a bogus message queue.

This test forces the `IDC_SLOW_OPERATION` button to start a timer:

```
TEST_(TestDialog, SetTimer)
{
    CButton slowButton = m_aDlg.GetDlgItem(IDC_SLOW_OPERATION);
    slowButton.SendMessage(BM_CLICK);
    BOOL thereWasaTimerToKill = m_aDlg.KillTimer(0);
    CPPUNIT_ASSERT(thereWasaTimerToKill);
}
```

The test detects the timer by successfully killing it. (Future refactors should replace the "Magic Number" `0` with a named constant.)

This code sets the timer, and moves the loop index into `ProjectDlg`'s member list. A third data member, for a dialog with so many features, is a small price to pay for event queue freedom:

```
    class
ProjectDlg:
    public CDialogImpl<ProjectDlg>
{
...
    BEGIN_MSG_MAP(ProjectDlg)
...
       COMMAND_ID_HANDLER(IDC_SLOW_OPERATION, OnSlowOperation)
       MESSAGE_HANDLER(WM_TIMER, OnTimer)
    END_MSG_MAP()

    LRESULT OnTimer(UINT, WPARAM, LPARAM, BOOL &);

    ProjectDlg(char const *xml):
        m_aCA(xml),
        m_index(0)
            {}
...
  private:
    CustomerAddress m_aCA;
    CString         m_fileName;
    int             m_index;
};
```

`OnSlowOperation()` will now work by checking if `m_index` is at the beginning or end of its progress. When no slow operation is under way the method starts it, otherwise it stops it. Here is an assertion-free test case to run a timer and watch the progress bar:

```
TEST_(TestDialog, SlowOperation)
{
    stuffLongUserList(m_aDlg);
    CButton slowButton = m_aDlg.GetDlgItem(IDC_SLOW_OPERATION);
```

```
        slowButton.SendMessage(BM_CLICK);
        revealFor("phlip");
    }

    RETEST_(TestDialog, TestSanskrit, SlowOperation)
...
        LRESULT
    ProjectDlg::OnSlowOperation(WORD, WORD, HWND, BOOL &)
    {
        CListBox aList = GetDlgItem(IDC_LIST_CUSTOMERS);
        int count(aList.GetCount());

        if (m_index > 0 && m_index < count)
            KillTimer(0);
        else
            {
            CProgressBarCtrl aBar = GetDlgItem(IDC_PROGRESS);
            aBar.SetRange(0, count);
            aBar.SetPos(0);
            m_index = 0;
            SetTimer(0, 0);
            }

        return 0;
    }

        LRESULT
    ProjectDlg::OnTimer(UINT, WPARAM, LPARAM, BOOL &)
    {
        Sleep(25);  //  put slow function here
        CProgressBarCtrl aBar = GetDlgItem(IDC_PROGRESS);
        aBar.SetPos(m_index);
        ++m_index;
        CListBox aList = GetDlgItem(IDC_LIST_CUSTOMERS);
        int count(aList.GetCount());
        return count > m_index;
    }
```

If `Sleep(25)` were instead a Logic Layer operation, that design would force it to be event driven. It could not make assumptions about the sequence of customers, or their index. It would only deal with the current customer, decoupled from other customer records, and from the GUI Layer.

That test case reveals a progress bar inching across our little dialog. Twice.

ImageMagick

Complex and intricate features need frequent review. To push down the cost of review, we will take pictures of our windows under test, and upload these to a gallery. So we will need a cheap and effective test-side system to manage images.

Our first effort takes pictures of *Temporary Visual Inspections*. Then we'll record animations of *Temporary Interactive Tests*.

To take a picture of a window and save the result as a GIF file, I wrote a utility called `screenCapture`. (The nifty ImageMagick tools cannot capture an MS Windows window, but they will soon do everything else to image files.) You may have a similar capture utility, or you might compile my `screenCapture.exe` from this book's online source.

The code inside `screenCapture.cpp` only illustrates low-level GDI functions, and not everyone could tolerate more of those, so rest assured I won't paste the whole thing in here.

To paint a window and then take its picture, `revealFor()` needs a mode that *Regulates the Event Queue* without blocking. The message `WM_PAINT`, and its attendant helper messages, will all run through the queue and paint their window. Then our test will continue.

That's why the *Broadband Feedback* Principle depends on fixtures generated from researching the *Regulate the Event Queue* Principle.

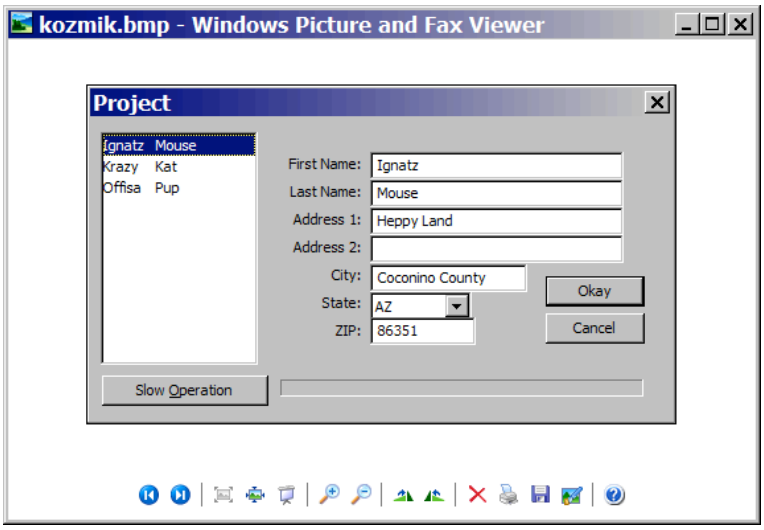
This code extends `revealFor()` to use `PeekMessage()`, the non-blocking version of `GetMessage()`. The `for` loop returns when the queue empties:

```
        class
    TestDialog:  public TestCase
    {
      public:
...
        bool revealFor(CString user, bool permanent = true);
...
    };

        bool
    TestDialog::revealFor(CString user, bool permanent)
    {
        if (getenv("USERNAME") != user)
            return false;


...
        for(;;)
            {
            BOOL bRet;

            if (permanent)
                {
                bRet = ::GetMessage(&msg, NULL, 0, 0);

                if (!bRet && msg.hwnd == m_aDlg)
                    break;
                }
            else
                {
                bRet = ::PeekMessage(&msg, NULL, 0, 0, PM_REMOVE);

                if (!bRet)
                    break;
                }
            ::DispatchMessage(&msg);
            }
           return true;
    }
```

These *Temporary Visual Inspections* paint our dialog, and take pictures of it:

```
    TEST_(TestDialog, ImageMagick)
    {
        if (revealFor("phlip", false))
            system("screenCapture kozmik.bmp");
    }

    RETEST_(TestDialog, TestSanskrit, ImageMagick)
```

`system()` treats an input string as a command line. It executes the target, and blocks until that target finishes. (Recall, at the beginning of the previous Case Study, we linked with `/SUBSYSTEM:CONSOLE`, for a seamless transcript of all our output, including the STDOUT from `system()`.)



After taking the picture, the test case runs `tearDown()`, which destroys this window. It only flickered on the screen, but that was long enough for `screenCapture` to preserve it for posterity. `screenCapture` targets the currently active window, whatever it is. If you run these kinds of tests in batches on your desktop in the background, do not view questionable material in the foreground.

We must upgrade this new feature into a reliable system. The first opportunity for improvement is the file name. I initially wrote "`kozmik.bmp`" for one basic reason: To make certain I would not think too hard, so early, about a good file name, nor search on my drive for the result. We must make the file unique to each test. Our test rig enforces unique test case names within an executable, so these names will become the output file names.

To name each BMP file after its test, the `TEST_()` macros will collect each test's name:

```
#define TEST_(suite, target)              \
    struct suite##target:  virtual suite \
    { virtual CStringA getName() { return #suite#target; } \
      void runCase(); }                   \
    a##suite##target;                     \
    void suite##target::runCase()

#define RETEST_(base, suite, target)     \
    struct base##suite##target:          \
        virtual suite,                   \
        virtual base##target  {          \
     CStringA getName() { return #base#suite#target; } \
     void setUp() { suite::setUp(); }    \
     void runCase() { base##target::runCase(); }  \
     void tearDown() { suite::tearDown(); }  \
    } a##base##suite##target;
```

Now use those names in the snapshot process:

```
    void
```

```
capture(CStringA name)
{
    CStringA str;
    str.Format("screenCapture %s.bmp", name);
    system(str);
}

TEST_(TestDialog, ImageMagick)
{
    if (revealFor("phlip", false))
        capture(getName());
}

RETEST_(TestDialog, TestSanskrit, ImageMagick)
```

That change generates two little BMP files:

The next step converts BMP files into GIF files—from flabby to portable.

ImageMagick, by John Cristy, is a sweet image processing suite. It's available at `http://www.imagemagick.org/`. Many GNU distributions bundle it, including CygWin and Linux.

Libraries are easier to add to test code than production code. ImageMagick runs from the command line, not our test code or production code.

Its `convert` command removes the flab:

```
    void
capture(CStringA name)
{
    CStringA str;
    str.Format("screenCapture %s.bmp", name);
    system(str);
    str.Format("convert %s.bmp %s.gif", name, name);
    system(str);
    remove(name + ".bmp"); //  clean up
}
```

(Warning: If ImageMagick's `convert` is not on your path, that command line could attempt to `convert` your hard drive's file system! Put a breakpoint on the `remove()` statement to inspect the test's console output and see if this happened. The wrong command might complain "`Invalid Parameter - TestDialogImageMagick.gif`". If you foolishly named your drive "`TestDialogImageMagick.gif`", you might get into even more trouble!

(BTW `convert` is naturally silent; `-verbose` should fix that.)

`capture()` runs…

```
convert TestDialogImageMagick.bmp TestDialogImageMagick.gif
```

…and `convert` detects the target file type from its extension, "`.gif`".

Animated GIFs

Our last feature to capture is animation. After writing a list of files, ImageMagick's `convert` will roll them all up into an animated GIF file. (Animated MNG files would be more modern, but neither Internet Explorer nor Windows XP support them. ImageMagick does.)

We need a *Temporary Interactive Test* that strobes our progress bar, taking a picture of it after each tick. Notice we decoupled the progress bar from its own loop statement to assist usability, robustness, an event-driven design, and tests. We *Regulated the Event Queue* to write the original `revealFor()` fixture. None of that research was proactive, looking ahead to *Broadband Feedback*. Because decoupling, event-driven designs, and event queue regulation are *Good Things*, this new kind of test is very easy to write:

```
    void
captureCell(CStringA name, int index)
{
    CStringA str;
    str.Format("screenCapture %s%03i.bmp", name, index);
    system(str);
}

TEST_(TestDialog, Animate)
{
    stuffLongUserList(m_aDlg, 50);
    CButton slowButton = m_aDlg.GetDlgItem(IDC_SLOW_OPERATION);
    slowButton.SendMessage(BM_CLICK);
    m_aDlg.KillTimer(0);  //  tests need fake timers

    for (int index(0);  index < 50;  ++index)
        {
        m_aDlg.SendMessage(WM_TIMER);

        if (revealFor("phlip", false))
            captureCell(getName(), index);
        }
    if (revealFor("phlip", false))
        animateCells(getName(), 50);
}
```

Before the call to `animateCells()`, `screenCapture.exe` filled the current folder with many numbered snapshots, like these:

```
...
```

Notice the `Format()` call uses `%03i`, giving the file names numbers with zero-padding: "`037`". (Research `printf()`, in various languages, to learn about these codes.) Our fixture doesn't require that extra esthetic touch. Folder explorers can sort by name to display the snapshots in chronological order (until the next function erases them all). Our fixture will simply regenerate the list of GIFs without reading the folder directory.

`animateCells()` puts all the BMPs into one animated GIF:

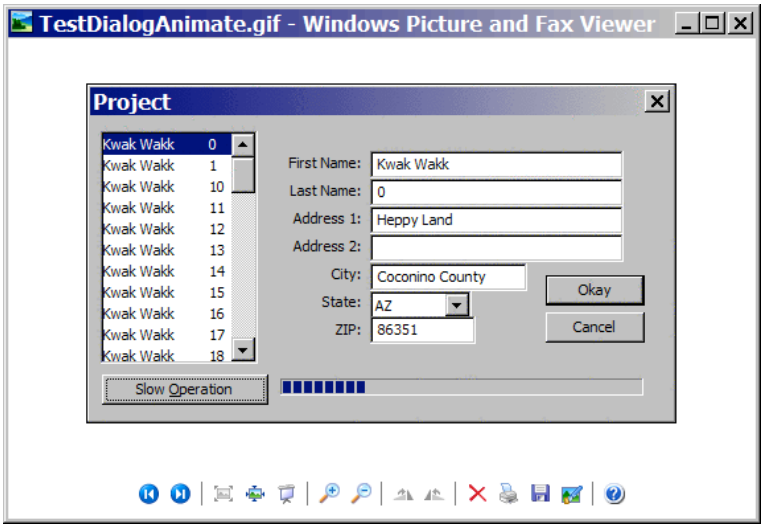
```
    void
animateCells(CStringA name, int max)
{

    CStringA cmd("convert ");
```

```
    for (int index(0);  index < max;  ++index)
        {
        CStringA str;
        str.Format("%s%03i.bmp ", name, index);
        cmd += str;
        }
    cmd += name + ".gif";
    system(cmd);

    for (int index(0);  index < max;  ++index)
        {
        CStringA str;
        str.Format("%s%03i.bmp ", name, index);
        remove(str);  //  delete intermediate files
        }

}
```

That builds a long command line…

```
convert TestDialogAnimate000.bmp TestDialogAnimate001.bmp …
                                                    TestDialogAnimate.gif
```

…and ImageMagick makes a movie:



The lowly progress bar crawls across that animated GIF (but you can't see that in print). Given a business reason to animate other controls, we could easily extend these fixtures to cover them.

That slow progress bar, and the number crunching inside `convert`, make for a very slow test. Such tests should move to your integration test suites, so they only run a few times a day, or at need.

Test Modes

Some tests run quick and quiet, to help engineers hit their test button during feature changes, as they drive software development's inner cycle. Some tests are slow and flashy, so we can take pictures of them. These pictures will help software development's outer cycle.

`Project.exe` currently only takes pictures if you edit `revealFor()` and recompile. We need it to trigger photography from these new command lines:

```
Project.exe --test files/testProject.xml /suite/case[2]
```

XML & XPath are back in the picture. This time, `files/testProject.xml` is a list of test resources, and `/suite/case[2]` points to one resource. A future version should test everything in the XML file when the command line contains no XPath.

`files/testProject.xml` contains:

```
<suite>
  <case
    locale="en" listIndex="0" first_name="Ignatz" last_name="Mouse"
    >Tests the first customer name, in the English locale</case>
  <case
    locale="sa" listIndex="1" first_name="Krazy" last_name="Kat"
    >Tests the second customer name, in the Sanskrit locale</case>
</suite>
```

A new kind of test requires new support fixtures. This new support fixture simply reads an XML file:

```
    typedef MSXML2::IXMLDOMNodePtr         pNode_t;
    typedef MSXML2::IXMLDOMNamedNodeMapPtr pMap_t;

    IXMLDOMDocument2Ptr
readXML(_bstr_t fileName)
{
    IXMLDOMDocument2Ptr pXML;
    GUARD(pXML.CreateInstance(L"Msxml2.DOMDocument"));
    pXML->setProperty("SelectionLanguage", "XPath");
    pXML->async = true;
    std::string xml = readFile(fileName);
    pXML->loadXML(xml.c_str());
    return pXML;
}
```

We saw something similar on page 172, but until now nothing re-used those abilities, so they were in no function of their own yet.

Code will soon query `pXML` for `/suite/case[2]`, and put the target node in `pNode`.

This next support method covers a “feature” in MS Windows. To prevent ambitious background programs from pushing windows up in front of foreground applications (interrupting users flow and redirecting their input events), MS changed the behavior of the `BringWindowToTop()` family of functions. They only go all the way to the top of a desktop if they are attached to the foreground thread of a foreground application.

This support function changes the behavior of `BringWindowToTop()` back to its Win95 equivalent. It works by attaching the window to the foreground thread of the foreground application:

```
    BOOL
forceWindowToTop(HWND windowToForce)
{
    BOOL  result = FALSE;
```

```
    //  subvert Win32 and push any window to the top
    //  (if popup ads can do it, we can do it!)

    //  find the topmost window and its thread

        HWND  topWindow = GetForegroundWindow();
        int   topThread = GetWindowThreadProcessId(topWindow, 0);

    //  make Win32 think our thread is in its session

        int   myThread  = GetCurrentThreadId();
        AttachThreadInput(myThread, topThread, TRUE);

    //  really put the window on top

        result = BringWindowToTop(windowToForce);
        assert(result);  //  our function is only for tests

    //  detach from the former top window's thread

        AttachThreadInput(myThread, topThread, FALSE);

    //  repaint

        InvalidateRect(windowToForce, NULL, TRUE);
        UpdateWindow(windowToForce);
        return result;
    }

        bool
    TestDialog::revealFor(CString user, bool permanent)
    {
        if (getenv("USERNAME") != user)
            return false;

        m_aDlg.ShowWindow(SW_SHOW);
        forceWindowToTop(m_aDlg);
…
    }
```

With our window reliably on top, unattended test batches, running in background processes, can reliably take pictures of windows during tests.

This next support fixture, for our sample test run, selects the second item in the list box, and checks that the first two edit fields now contain "`Krazy`" and "`Kat`".

We report "`pass`" to the command line, because copious other low-level tests constrain this feature!

```
        void
    testTargetListboxItem(ProjectDlg & aDlg, pNode_t & pNode)
    {
        pMap_t  map    = pNode->attributes;
        CListBox aList = aDlg.GetDlgItem(IDC_LIST_CUSTOMERS);
       _bstr_t listIndex  = map->getNamedItem("listIndex")->text;
        aList.SetCurSel( _ttoi(listIndex) );
        BOOL b(false);
        aDlg.OnSelChange(0,0,0,b);  //  Weak User Simulation
```

```
//  test the list box populates the edit fields correctly

   _bstr_t first_name = map->getNamedItem("first_name")->text;
   _bstr_t  last_name = map->getNamedItem( "last_name")->text;

    if ( aDlg.getText(IDC_EDIT_FIRST_NAME) == LPCWSTR(first_name) &&
         aDlg.getText(IDC_EDIT_LAST_NAME ) == LPCWSTR(last_name) )
        cout << "pass";
    else
        cout << "fail";
}
```

The test case will soon use the `RETEST_()` system to optionally take pictures of either the `LANG_ENGLISH` or `LANG_SANSKRIT` skin. Until now, test cases could not easily query which locale they occupied. This new member variable remembers:

```
    class
TestDialog:  public TestCase
{
  public:
    ProjectDlg m_aDlg;
    int        m_myLocale;
…
    TestDialog():  m_aDlg(xml), m_myLocale(LANG_ENGLISH)  {}
};
```

Note that variable resembles a public data member! Fear not; it is private to the file `Project_tests.cpp`. C++ programs need not arbitrarily put every class into a `.h` file.

The derived test fixture bonds its cases with the Sanskrit locale:

```
    struct
TestSanskrit:  virtual TestDialog
{

    void
setUp()
    {
    m_myLocale = LANG_SANSKRIT;
    WORD sanskrit(MAKELANGID(LANG_SANSKRIT, SUBLANG_DEFAULT));
    LCID lcid(MAKELCID(sanskrit, SORT_DEFAULT));
    ::SetThreadLocale(lcid);
    TestDialog::setUp();
    }
…

};
```

This function captures the current screen and moves it to my Web server's working folder:

```
    CStringA
captureAndSendToServer(CStringA gifName)
{
    capture(gifName);  //  say "cheese"
    gifName += ".gif";
    CStringA target = "C:/phlip/WebSite/files/" + gifName;

//  send the file to the server's source folder
```

```
    BOOL moved = MoveFileExA( gifName, target,
                              MOVEFILE_REPLACE_EXISTING );
    assert(moved);
    return gifName;
}
```

The code that puts them all together is a little rough. Naturally, if a project grew many of these fixtures, they could merge their common features better.

```
TEST_(TestDialog, BroadbandFeedback)
{

// Syntax: Project.exe --test <file.xml> <xpath>

    if (__argc < 4)
        return;

    _bstr_t fileName (__argv[2]);
    _bstr_t xPath    (__argv[3]);

    IXMLDOMDocument2Ptr pXML = readXML(fileName);

//  find the selected test case

    pNode_t pNode  = pXML->selectSingleNode(xPath);
    pMap_t  map    = pNode->attributes;
    _bstr_t locale = map->getNamedItem("locale")->text;

//  match the locale to this particular test
//  case (see RETEST_() below)

if ((locale == _bstr_t("sa")) == (m_myLocale == LANG_ENGLISH))
        return;

    testTargetListboxItem(m_aDlg, pNode);

//  paint the dialog, take a picture of it,
//  and send this to the Web server

    if (revealFor("phlip", false))
        {
        CStringA gifName = captureAndSendToServer(getName());
        cout << "!http:files/" + gifName;
        }
    cout << endl;

}

RETEST_(TestDialog, TestSanskrit, BroadbandFeedback)
```

That code emits only this to the console:

```
pass!http:files/TestDialogTestSanskritBroadbandFeedback.gif
```

A test runner will call this fixture and sample that from the command line, to learn its single case result and the result's output location, relative to a Web server.

One final point: The `revealFor()` should take a user-name account on your test server. Without it, the test will run but capture an image.

Now to review what all that did. Given a command line with an XML file and an XPath pointer, that traversed the XML to this test resource:

```
  <case locale="sa" listIndex="1" first_name="Krazy">Kat</case>
```

That told it to select the skin with the Sanskrit locale (ISO 693-1 code "`sa`"), and then select the second list box item. Then the test fixture checked that the "first name" edit field said "`Krazy`" and the "last name" edit field said "`Kat`":

(Localizing the sample data, "Krazy Kat", to Sanskrit is left as an exercise for the reader. There's more than one way to skin a Kat.)

The test fixture took a picture of those effects, and moved it one of my Web server's folders.

The test resource manipulated the list box by transmitting a new index, `listIndex`. Some resources should transmit explicit commands to named controls, like this:

```
<suite>
   <case>
    <action control="Customer List" event="select" argument="1" />
    <action control="City" event="edit" argument="Lithoid Mittens" />
…
  </case>
</suite>
```

The more a test fixture manipulates a GUI's controls generically, the more it resembles a GUI scripting layer, such as MS ActiveX Script. Such layers permit users to record a sequence of inputs and store them into a single macro command. Some projects install these layers explicitly to test.

Conclusion

This Case Study pushed the limits of testing, and illustrated many common GUI concepts—low-level graphics, localization, and animation—while investigating just a few various and diverse technical solutions.

Then we developed a single, platform-neutral system to constrain all of them. The code that shells to ImageMagick could easily port to any language, on any GNU-enabled platform. Fixtures must aggressively Regulate your Event Queue, permitting reliable screen captures.

After creating those GIF files, the next natural step is upload them to a Web site, so Web pages can host them for rapid, distributed browsing. The end of the last Case Study, *The Web*, connects these dots.

But first, we're going to have a little fun.

Chapter 10: *Embedded GNU C++ Mophun™ Games*

I sought an embedded platform for a while, to address some important industry sectors. But I made the mistake of asking a forum called "C2 Wiki" what "WAP" is, and only learned that it was a proprietary vehicle for popup™ advertisements©, and a tool of The Man®, etc, etc.

Then my family gave me a tiny Sony Ericsson T226 cell phone, with a 101x80 pixel 512-color Liquid Crystal Display. These newfangled contraptions exceed many specifications of last decade's workstations.

This Case Study begins to write a simple game for it:

Mophun Learner Test

The second thing I did with that new phone was switch the ringing sound from "The Ride of the Valkyries on Speed" or whatever to an audio sample of a real phone ringing with an oscillating hammer between two bells.

The readers can easily guess the first thing I did with it:

```
#include <vmgp.h>
#include "resource.h"

    VMGPFONT FontA;

    void
test_vPrint()
{
    FontA.width     = 4;
    FontA.height    = 6;
    FontA.bpp       = 1;
    FontA.palindex  = 1;
    FontA.chartbl   = CONSOLEFONT + 16;
    FontA.fontdata  = CONSOLEFONT + 16 + 256;
    vSetActiveFont(&FontA);

    vClearScreen(40);
    vSetForeColor(255);
```

```
    int x(8);
    uint16_t background = vGetPixel(x, x);
    vPrint(MODE_TRANS, 2, 8, "HELLO WORLD");
    vFlipScreen(1);

    uint16_t pixel = vGetPixel(x, x);

    if (pixel == background)
        DbgPrintf
            (
            "pixel(%i,%i) is still background color %x\n",
            x, x,
            pixel
            );
}

    int
main()
{
    test_vPrint();
    return 0;
}
```

That Learner Test runs in a cross-compiling environment, based on GNU g++, supporting a gaming library with the trade name Mophun, technical name VMGP (Virtual Mobil Gaming Platform), by Antony Hartley and Anders Norlander of Synergenix Interactive AB.

The environment tests programs in software using a full-featured emulator. This reads configuration files to emulate any of a long list of cell-phone model numbers—T226, T230, T610, T616, etc. My test exercised these functions in the emulator:

- `vSetActiveFont()` selects a font
- `vClearScreen()` erases all pixels
- `vGetPixel()` samples a pixel, for our test
- `vPrint()` writes ("HELLO WORLD") in a tiny font
- `vGetPixel()` samples that same pixel again
- `vFlipScreen()` refreshes the hardware screen.

If `vPrint()` worked (and if the font geometry did not change), then the pixel should change color from dark green to white. If it did not, we send a complaint into the debugging trace channel, `DbgPrintf()` (note the missing `v`). The Mophun.exe cell phone emulator writes this into its output debugging window.

And we are about to throw the test away. It's a just Learner Test—a snip of one of the tutorial sources, converted into test format, with a primeval assertion.

The cross-compiler supports features from many C languages. But this freedom prevents `namespaces`. Library identifiers follow the Registered Package Prefix technique from *Large Scale C++ Software Design*—they all begin with a `v`. But we obviously won't go large scale here.

Sane Embedded Subset

Programmers beginning in the C languages often emulate their elders by spending time making sure all code appears as fast and small as possible, sacrificing clarity. For example, they might multiply `x` by `4` cleverly, using `x << 2`. Compilers have such optimizations well within

reach, and programmers can't guess a CPU won't evaluate `x * 4` faster than `x << 2`. Compilers will usually know that.

Then elders remind programmers that "premature optimization" wastes time and degrades designs. Programmers can never predict where their slow or fat spots will be as well as profiling tools can detect these, after a design matures. Then the fix is easy. Programmers should instead follow a Sane Subset that propels development. It must only suggest optimizations impossible for compilers, and which can't require thinking about trade-offs. For example, large objects should pass not by value—`foo(SimCity aCity)`—for example, but by reference: `foo(SimCity const &aCity)`.

Embedded work changes your Sane Subset. My cell phone's manual recommends each game use a 60-kilobyte sandbox. We must even wonder which cell phone models use what Application Programming Interfaces versions, and which functions developed late.

In the above code

Consoles on full-featured platforms are both display devices and streams. Most tests on consoles need only test their streams. Cell phones can't output text as streams, only as graphical operations. Testing here will be interesting. But at least we can always sample pixels, right?

When I switch the Mophun Emulator from the T610 emulation profile to the T226 emulation profile, my first test fails with an (emulated) hardware error: "`Unresolved symbol access: vGetPixel`". A T226 uses Mophun version 1.10; `vGetPixel()` needs versions >=1.50. It seems somebody did not start their library with a test…

Versions of both the emulator and the cell phone reject programs that call non-existing functions, so we must sometimes conditionally compile away all `vGetPixel()` calls. We should compile three versions—v1.50 with tests, v1.10 with tests, and v1.10 without tests—the release build. My main disappointment is I won't be able to run *all* the tests inside my actual cell phone!

Sprights

Game platforms always provide a system to draw a figure and move it around over a background. Here is a `SPRITE` data record for Mophun:

```
SPRITE sprite_face =
{
    0,       //  palindex - a Palette offset
VCAPS_RGB332, //  format - of pixel data
    4, 4,    //  centerx, ~y - the hotspot
    8, 8     //  width, height - a square here
};

uint8_t raw_face[] =  //  the pixel data
{
      0,   0,0xE0,0xE0,0xE0,0xE0,   0,   0,
      0,0xE0,   0,0xE0,0xE0,   0,0xE0,   0,
   0xE0,0xE0,   0,0xE0,0xE0,   0,0xE0,0xE0,
   0xE0,0xE0,   0,0xE0,0xE0,   0,0xE0,0xE0,
   0xE0,0xE0,0xE0,0xE0,0xE0,0xE0,0xE0,0xE0,
   0xE0,0xE0,   0,   0,   0,   0,0xE0,0xE0,
      0,0xE0,0xE0,0xE0,0xE0,0xE0,0xE0,   0,
      0,   0,0xE0,0xE0,0xE0,0xE0,   0,   0,
};
```

One always defines a sprite by defining its header `SPRITE` structure, then defining an array of pixels, of the correct size, immediately following it in memory.

If the `SPRITE` structure did not depend on implementation-defined C features, such as the padding inside and between sequentially declared aggregates in global storage, it would be a "variable sized structure", to allocate only on the heap. In embedded land, however, sometimes the implementation specifies zero padding. We can put two data aggregates "next to each other" in global storage, and users of `sprite_circle` will simply know that the `SPRITE`'s raster image always begins at the address `(&sprite_face + 1)`. Our Sane Subset is shifting!

Even though target phones support 9 or more bits of color, the simplest and briefest bitmap format available, `VCAPS_RGB332`, uses 8 bits per pixel, providing 256 colors.

Squint at the `0xE0` pixels. They form the surprised face of the intrepid hero of our next adventure.

To prove she or he is worthy of entering (whatever) game I'm about to write, here's the first test the adventurer must pass:

```
#define ASSERT_(x_) if (!(x_)) \
         DbgPrintf(#x_": %i\n", (x_))

#define ASSERT_EQUAL(x_,y_) if ((x_) != (y_)) \
         DbgPrintf("failed: " #x_": %i "       \
                             #y_": %i \n", (x_), (y_))

    void
test_moveAsprite()
{
    uint32_t q = vSpriteInit(1);
    ASSERT_(q);

    int x = 15;

    for (; x < 50;  ++x)
        {
        vSpriteSet(0, &sprite_face, x, x);
        vUpdateSpriteMap(); // Draw background & sprites
        vFlipScreen(1);
        }
    uint16_t shouldBeRed = vGetPixel(49, 49);
    ASSERT_EQUAL(0x7C00, shouldBeRed);
    vSpriteDispose();  //  free the slot array
}

    int
main()
{
    test_console();
    test_moveAsprite();
    return 0;
}
```

This test demonstrates the `SPRITE` library in action. Its C-style API manages an array of `SPRITE` addresses in "slots". That means internally it only stores an array of pointers to your data structures. By contrast, libraries on full-featured platforms decouple their own storage systems from your application's memory. MS Windows' `HANDLE` architecture provides this capacity, generically for all managed objects. When my code passes the address of a block of

data into an OS function, after that function exits, ownership passes back to my code. The OS copies the data it needs and returns only `HANDLE`s, not pointers, to the objects that it manages. MS Windows usually does not remember the addresses of structures I pass to it.

Embedded systems have no room for all this paranoia, secret handshakes, and redundant bookkeeping. Embedded systems depend on your code to behave. Tests and inline assertions should verify things that a fatter OS would have verified for us.

Between the call to `vSpriteSet()` and the call to `vUpdateSpriteMap()`, if I call `memset(raw_face, 0, sizeof raw_face)` during the test, then the face will disappear from the emulator—and the test will fail.

Our little test calls `vSpriteSet()` and `vUpdateSpriteMap()`, right next to each other, so this effect seems obvious. But the "slot" architecture intends to permit arbitrarily complex code to move the `SPRITE`s around, and one `Update` call to rule them all and bind them. If our source becomes complicated, as we add new features, then mistakes would be less obvious.

VMPG does not copy my sprite into its memory, it simply remembers the address of the one I provided. My code can get clever, but not so clever that it abuses any objects after I pass their addresses into the API.

Our new, primitive macro, `ASSERT_EQUAL()`, at failure time, only reports "`failed: 0xE0: 224 shouldBeRed: 0`". It can only accept integers; it does not print `__FILE__` or `__LINE__`, and it does not invoke a debugger. Because the Model View Controller Case Study already showed assertions with more features (see page 170), and because adding them to a `printf()`-style trace statement would create horrid cruft without hope of elegance, we will investigate improving our assertion's *Fault Navigation* only as tests get more complex.

The assertion reference color in `ASSERT_EQUAL(0x7C00, shouldBeRed)` is not `0xE0` because those were pixels interleaved into bit planes. `vGetPixel()` returned a vertical sample through these.

Here's a detail of the Mophun.exe emulator, set to T610 mode, with our tiny little smiley sprite in its final location:

The emulator leaves its simulated screen in its final state after emulating. Our *Temporary Visual Inspection* Principle appears automatically—for the last test that happens to run. Placing the test under current development at the bottom of our main test list, where we know it will run last, leverages this effect to our benefit. That won't significantly violate the Test Isolation principle.

Our adventurer is afraid of the dark. Let them pick up something to make them feel secure:

```
    SPRITE sprite_gun =
    {
        0,
    VCAPS_RGB332,
         3,  3,
        13,  9,
    };

    uint8_t raw_gun[] =  //  the pixel data
    {
   0,   0,0x0E,   0,   0,   0,   0,   0,   0,   0,   0,0x0E,
   0,   0,0x0E,0x0E,0x0E,0x0E,0x0E,0x0E,0x0E,0x0E,0x0E,0x0E,
   0,   0,   0,0x0E,0x0E,0x0E,0x0E,0x0E,0x0E,0x0E,0x0E,0x0E,
   0,   0,0x0E,0x0E,0x0E,   0,   0,   0,   0,   0,   0,   0,
0x0E,0x0E,0x0E,0x0E,   0,   0,   0,   0,   0,   0,   0,   0,
```

```
0x0E,0x0E,0x0E,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0x0E,0x0E,0x0E,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0x0E,0x0E,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0x0E,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
    };

        void
    test_pickUpGun()
    {
        uint32_t q = vSpriteInit(2);
        ASSERT_(q);

        int x = 15;
        vSpriteSet(1, &sprite_face, x, x);

      //  plant useful gun in adventurer's path

        vSpriteSet(0, &sprite_gun, 23, 23);
        vUpdateSpriteMap();
        vFlipScreen(1);


        int blue = vGetPixel(23,23);  //  find the blue gun
        ASSERT_EQUAL(0x1b4, blue);

        for (x = 16; x < 50;  ++x)
            {
            vSpriteSet(1, &sprite_face, x, x);
            vUpdateSpriteMap();
            vFlipScreen(1);
            }

      //  moving over a useful object should pick it up

        blue = vGetPixel(23,23);

      //  black, 'cause gun shouldn't be there

        ASSERT_EQUAL(0, blue);
        vSpriteDispose();
    }

        int
    main()
    {
        test_console();
        test_moveAsprite();
        test_pickUpGun();
        return 0;
    }
```

That test fails, and to fix it we need a collision detector. I could write a simple one, or use the one built into VMGP. If I wrote a simple one, it would trivially detect when two rectangles overlap. But this detail shows that the `0` pixels in our `SPRITE`s' rasters are transparent:

(The face is on top of the gun because we put `sprite_face` into the slot with the highest index. When objects grow around this system, the tests should enforce an invariant that the Ego sprite occupies the "last" slot.)

If rectangles around sprites detected collisions, then the user would feel like getting near the gun picks it up, and this would damage suspension of disbelief. So VMGP's collision detector is the best way to mask in the exact colored shape of each sprite:

```
…
        int gunX = 23;
        int gunY = 23;
        vSpriteSet(0, &sprite_gun, gunX, gunY);
        vUpdateSpriteMap();
        vFlipScreen(1);

        int blue = vGetPixel(23,23);
        ASSERT_EQUAL(0x1b4, blue);

        for (x = 16; x < 50;  ++x)
            {
            vSpriteSet(0, &sprite_gun, gunX, gunY);
            vSpriteSet(1, &sprite_face, x, x);

            int16_t hit = vSpriteCollision(1, 0, 0);

            if (-1 != hit)
                {
                gunX = x;  //  you are coming
                gunY = x;  //    with me
                }
            vUpdateSpriteMap();
            vFlipScreen(1);
            }
        blue = vGetPixel(23,23);
        ASSERT_EQUAL(0, blue);
…
```

The test passes; the animation (not show) looks like the adventurer running over, picking up, & carrying the gun, and the implementation contains more lies than truths.

In theory, all of that duplicated behavior should go into test fixtures and production objects. But if we perform the Extract Method Refactor on the above code, the lies could travel into new, elaborate classes that might then apparently resist fixing them.

The solution, as always, is to start with the simplest possible code fixes, and with the tests. They frequently check a pixel's color, so we should fixturize that:

```
#define ASSERT_COLOR(x_,y_,c_)                    \
    if ((c_) != vGetPixel((x_), (y_)))             \
     DbgPrintf( "wrong color at(%i,%i): 0x%X != 0x%X\n", \
                              (x_),(y_),(c_),       \
                             vGetPixel((x_), (y_)))
```

At failure time it prints:

```
wrong color at(22,22): 0x1B4 != 0x0
```

We might call this the “Extract Macro Refactor”. But every single pesky `vGetPixel()` just disappeared into it:

```
    ASSERT_COLOR(49, 49, 0x7C00);
…
    ASSERT_COLOR(23, 23, 0x1b4);
…
    ASSERT_COLOR(23, 23, 0);
```

That will simplify the conditional compilations required to deploy to multiple API versions!

Looking at our object model, we have “latent objects” in all places where we see “two or more variables that always appear together”:

```
        vSpriteSet(0, &sprite_gun, gunX, gunY);
…
        vSpriteSet(1, &sprite_face, x, x);
```

The `sprite_face` uses `x`, `x` because the test moves the sprite down an absurdly convenient diagonal line. The principle Grind it ‘Till you Find it implies we should make similar things more similar before removing duplication:

```
        vSpriteSet(0, &sprite_gun, gunX, gunY);
…
        vSpriteSet(1, &sprite_face, faceX, faceY);
```

The tests now use those variable names (not shown), and they pass.

Now wrap the C `struct SPRITE` in a `class`. Its objects’ identities and lifespans will parallel the internal objects:

```
    class
Spright
{
  public:
    SPRITE  &m_aSprite;
    int16_t  m_exx;
    int16_t  m_why;
    int16_t  m_zee;

Spright( SPRITE  &aSprite,
         int16_t  exx,
         int16_t  why,
         int16_t  zee ):
        m_aSprite(aSprite),
        m_exx    (exx),
        m_why    (why),
        m_zee    (zee)
            {}

    void
appear()
    {
    vSpriteSet(m_zee, &m_aSprite, m_exx, m_why);
    }
};
```

As a style rule, there's a reason those data members are `public`. But the reason is not the usual one "because the design is emerging". The reason is that class lives down in the implementation file `Mutato.cpp`, not up in a header. The entire class is private, so it would already encapsulate relative to other modules (if there were any).

But because the design is, indeed, emerging, as those members naturally become private we will declare them `private`. And that, emergently, will make moving `Spright` to a header file easier when the time comes.

Along the way, the X and Y types upgraded from a casual `int` to a formal `int16_t`. Nothing uses the new `Spright` class yet; first we check that it can compile, and that it does not interfere with the tests, even thought they don't use it yet.

And the link to `aSprite` changed from a pointer to a reference. Always use references without a reason to use pointers. During a run, `aSprite` currently never re-seats, and never references `NULL`. Ironically, the advanced language C++ gives us the correct semantic way to express the primitive embedded C API's use of objects in our memory.

After the tests pass, we replace the latent `gun` object with a real `Spright` object:

```
    void
test_pickUpGun()
{
    uint32_t q = vSpriteInit(2);
    ASSERT_(q);

    int faceX = 15;
    int faceY = 15;
    vSpriteSet(1, &sprite_face, faceX, faceY);

    Spright gun(sprite_gun, 23, 23, 0);
    gun.appear();
    vUpdateSpriteMap();
    vFlipScreen(1);

    ASSERT_COLOR(23, 23, 0x1B5);

    for (faceX = 16; faceX < 50;  ++faceX, ++faceY)
        {
        gun.appear();
        vSpriteSet(1, &sprite_face, faceX, faceY);

        int16_t hit = vSpriteCollision(1, 0, 0);

        if (-1 != hit)
            {
            gun.m_exx = faceX;
            gun.m_why = faceY;
            }
        vUpdateSpriteMap();
        vFlipScreen(1);
        }
    ASSERT_COLOR(23, 23, 0);
    ASSERT_EQUAL(gun.m_exx, faceX - 1);
    ASSERT_EQUAL(gun.m_why, faceY - 1);
    vSpriteDispose();
}
```

The VMGP API efficiently coupled the concept of a `SPRITE`'s location to that of its entry in a slot array. Our new `Spright` class should seek to abstract the `SPRITE`'s position from its

display mechanics. And wrapping the `spright_face` in a new `SPRITE` object makes a new set of duplicated behaviors more obvious:

```
…
        Spright gun(sprite_gun,  23, 23, 0);
        Spright ego(sprite_face, 15, 15, 1);
        gun.appear();
        vUpdateSpriteMap();
        vFlipScreen(1);

        ASSERT_COLOR(23, 23, 0x1B5);

        for ( ego.m_exx = 16;
              ego.m_exx < 50;
            ++ego.m_exx,
            ++ego.m_why )
            {
            gun.appear();
            ego.appear();

            int16_t hit = vSpriteCollision(1, 0, 0);

            if (-1 != hit)
                {
                gun.m_exx = ego.m_exx;
                gun.m_why = ego.m_why;
                }
            vUpdateSpriteMap();
            vFlipScreen(1);
            }
        ASSERT_COLOR(23, 23, 0);
        ASSERT_EQUAL(gun.m_exx, ego.m_exx - 1);
        ASSERT_EQUAL(gun.m_why, ego.m_why - 1);
…
```

My Duplication Detector sees this too often:

```
m_exx
m_why
```

The latent object here is a `Point` class. We create one, and replace `Spright::m_exx` and `Spright::m_why` with `Point Spright::at`.

We didn't use `m_at` because the little `m_` warts got too busy-looking. But I use the otherwise inane identifiers "`exx`" and "`why`" because the rule, "Use complete pronounceable names for identifiers," conflicts with another rule, "Use common math symbols, no matter how arbitrary." That conflicts with another rule, "Don't use single letters for important variables." One of those rules had to give.

Now that our new class has state, give it the behaviors that clients need from it. We overload `Point`'s operators for the first few operations we meet:

```
    class
Point
{
  public:
    int16_t exx;
    int16_t why;
```

```
    Point( int16_t exx_ = 0,
           int16_t why_ = 0 ):
        exx(exx_),
        why(why_)
            {}

        Point const &
    operator=(int x)
    {
        exx = why = x;
        return *this;
    }

        Point
    operator-(int x)
    {
        return Point(exx - x, why - x);
    }
};

    class
Spright
{
  public:
    SPRITE  &m_aSprite;
    Point    at;
    int16_t  m_zee;

Spright( SPRITE  &aSprite,
         Point    at_,
         int16_t  zee ):
        m_aSprite(aSprite),
               at(at_),
            m_zee(zee)
            {}

    void
appear()
    {
    vSpriteSet(m_zee, &m_aSprite, at.exx, at.why);
    }
};

#define ASSERT_SAME_POINT(p1,p2) \
            ASSERT_EQUAL((p1).exx, (p2).exx); \
            ASSERT_EQUAL((p1).why, (p2).why)

    void
test_pickUpGun()
{
    uint32_t q = vSpriteInit(2);
    ASSERT_(q);

    Spright gun(sprite_gun,  Point(23, 23), 0);
    Spright ego(sprite_face, Point(15, 15), 1);
    gun.appear();
    vUpdateSpriteMap();
    vFlipScreen(1);

    ASSERT_COLOR(23, 23, 0x1B5);
```

```
    for ( ego.at = 16;
          ego.at.exx < 50;
        ++ego.at.exx,
        ++ego.at.why )
        {
        gun.appear();
        ego.appear();

        int16_t hit = vSpriteCollision(1, 0, 0);

        if (-1 != hit)
            gun.at = ego.at;

        vUpdateSpriteMap();
        vFlipScreen(1);
        }
    ASSERT_COLOR(23, 23, 0);
    ASSERT_SAME_POINT(gun.at, ego.at - 1);
    vSpriteDispose();
}
```

As usual, when mathematics gets in the way of programming, math loses. Because our `Point` class must only have features the calling code needs, it does things that are less than mathematically coherent. `operator=(int)` and `operator-(int)` assign both members. They constrain our `Point`s to occupy the single Cartesian diagonal where `exx == why`. This is a by-product of how cheap and simple my first test was. As more tests teach `Point`s to do more things, incremental pragmatic changes will polish these logical irritations.

Reviewing the code, we find two remaining problems. The code that "picks up" the `gun` is a hack. It assumes the collider is the `gun`, then the `ego` "carries" the `gun` by constantly recolliding with it. When new code provides, say, a useless `rock` object, colliding with it would arbitrarily carry it!

And the code around `vSpriteInit()` and `vSpriteDispose()` duplicates across the two tests. That's now the simplest problem to fix here:

```
    class
Spright
{
  public:
    SPRITE     &m_aSprite;
    Point       at;
    int16_t     m_zee;
    static int  s_count;
    static bool s_initialized;

Spright( SPRITE &aSprite, Point at_,  int16_t  zee ):
        m_aSprite(aSprite), at(at_), m_zee(zee)
    {
    ++s_count;
    }

~Spright()
    {
    --s_count;

    if (0 == s_count && s_initialized)
        {
        vSpriteDispose();
```

```
        s_initialized = false;
        }
    }

    void
appear()
    {
    if (!s_initialized)
        {
        uint32_t q = vSpriteInit(s_count);
        ASSERT_(q);
        s_initialized = true;
        }
    vSpriteSet(m_zee, &m_aSprite, at.exx, at.why);
    }
};

    int  Spright::s_count = 0;
    bool Spright::s_initialized = false;
```

On a full-featured platform, we would need to worry about the possibility of more than one screen, or more than one `SPRITE` map. Here, we can assume that a `static` variable can reliably count the `SPRITE`s for our library's single `SPRITE` map.

The new class behaviors makes us remove the equivalent calls to `vSpriteInit()` and `vSpriteDispose()` from the calling code. Conceptually, `SPRITE`s now own and manage their `SPRITE` map.

However, the class does not work if we call `Spright::appear()`, then construct another `Spright`, then call its `appear()` method. Tough.

Don't Let Sleeping Goblins Lie

Here's a green sleeping goblin:

```
SPRITE sprite_goblin_asleep =
{
    0,
VCAPS_RGB332, //  3 bits red, 3 bits green, 2 bits blue
    4, 4,
    8, 8
};

#define c332(r,g,b) ( ( (7 & (r)) << 5 ) | \
                      ( (7 & (g)) << 2 ) | \
                        (3 & (b)) )

#define GREN c332(0,5,0)

uint8_t raw_goblin_asleep[] =
{
      0,   0,GREN,GREN,GREN,GREN,   0,   0,
      0,GREN,GREN,GREN,GREN,GREN,GREN,   0,
   GREN,   0,   0,GREN,GREN,   0,   0,GREN,
   GREN,GREN,GREN,GREN,GREN,GREN,GREN,GREN,
   GREN,GREN,GREN,GREN,GREN,GREN,GREN,GREN,
   GREN,GREN,GREN,   0,   0,GREN,GREN,GREN,
      0,GREN,GREN,GREN,GREN,GREN,GREN,   0,
      0,   0,GREN,GREN,GREN,GREN,   0,   0,
```

```
};
```

Though it's just asleep, not attacking, that Goblin might have Weapons of Mass Destruction or something in its pocket, so we must take no chances. This test shows our hero leaping at the gun, aiming at the Goblin, and shooting it:

```
#define db(x_) DbgPrintf(#x_": %i\n", x_)
#define dbx(x_) DbgPrintf(#x_": %X\n", x_)

    void
test_shootGoblin()
{

  //  inanimate objects always construct first;
  //  ego always last

    Spright    gun(sprite_gun,           Point(23, 23), 0);
    Spright goblin(sprite_goblin_asleep, Point(70, 60), 1);
    Spright    ego(sprite_face,          Point(15, 15), 2);

  //  jump ego to gun (via cheating):

    ego.at = gun.at;
    ego.pickUp(gun);

       gun.appear();
    goblin.appear();
       ego.appear();

    vUpdateSpriteMap();
    vFlipScreen(1);

    for (int x = 0; x < 6; ++x)
        for (int y = 0; y < 6; ++y)
            {
            db(x);
            db(y);
            dbx(vGetPixel(ego.at.exx + x, ego.at.why + y));
            }
}
```

Actually, the test does not do all that. After writing the previous test, the hero can't pick up the gun. To write this test, both the hero and I need to figure out, by printing out pixel colors, where the gun is!

This test will jump the hero onto the gun, call `pickUp()`, move the hero to aim at the sleeping goblin, and at shoot it. After this test passes, duplication removal with the last test will pull the collision detection system and `pickUp()` together. Both tests will get shorter, and the `Spright` class will grow closer to something we can use inside a real user-interface event message pump.

Before that happens, we need to test whether pixels near the hero are gun-colored. But to write the prior tests, I did not calculate in my head all the offsets involved. (Recall that a `SPRITE`'s image draws over its XY location offset by the hotspot values of `centerx` and

centery.) The above code scans a 6x6 pixel region to the southeast of ego's hotspot. When that proto-test runs, we find gun color (0x1B5) at an offset of (ego.at.exx + 0, ego.at.why + 1).

Our tests should eventually grow fixtures that seek pixels intelligently based on Spright::at values. Also, as vGetPixel() tests more primitive Spright methods into existence, the tests that create new features can assert them based on side effects of these primitive methods. All testage will lead back to vGetPixel(), but not all test cases will call it.

This test shows our hero leaping at the gun, aiming at the Goblin, and shooting it:

```
        class
    Spright
    {
      public:
...
        Spright   * m_pHolding;

    Spright( SPRITE &aSprite, Point at_,  int16_t  zee ):
            m_aSprite(aSprite), at(at_), m_zee(zee),
            m_pHolding(NULL)
...
    void pickUp(Spright &gun);
    };
...
        void
    Spright::pickUp(Spright &gun)
    {
        m_pHolding = &gun;
    }

        void
    test_shootGoblin()
    {

      //  inanimate objects always construct first;
      //  ego always last

        Spright    gun(sprite_gun,           Point(23, 23), 0);
        Spright goblin(sprite_goblin_asleep, Point(70, 60), 1);
        Spright    ego(sprite_face,          Point(15, 15), 2);

      //  jump ego to gun (via cheating):

        ego.at = gun.at;
        ego.pickUp(gun);

           gun.appear();
        goblin.appear();
           ego.appear();

        vUpdateSpriteMap();
        vFlipScreen(1);

        ASSERT_COLOR(ego.at.exx + 0, ego.at.why + 1, 0x1B5);

      //  jump ego to aim at goblin

        ego.at.exx = 50;
```

```
    ego.at.why = goblin.at.why;

  //  paint

       gun.appear();
    goblin.appear();
       ego.appear();

    vUpdateSpriteMap();
    vFlipScreen(1);
    ASSERT_COLOR(ego.at.exx + 0, ego.at.why + 1, 0x1B5);

}
```

It fails; our hero dropped his gun. (You couldn't make this stuff up!)

Our tests now also duplicate a lot. This heroic action is taking a while. Fortunately, our puttering around has not awakened that Goblin yet…

This new code helps the hero keep a grip on the `gun`:

```
    void
appear()
    {
    if (!s_initialized)
        {
        uint32_t q = vSpriteInit(s_count);
        ASSERT_(q);
        s_initialized = true;
        }
    if (m_pHolding)
        {
        m_pHolding->at = at;
        m_pHolding->appear();
        }
    vSpriteSet(m_zee, &m_aSprite, at.exx, at.why);
    }
```

It also makes one of the calls to `gun.appear()` physically redundant. But if the function it calls, `vSpriteSet()`, only updates internal variables, then `gun.appear()` might be efficient enough for us to abuse.

Here are some frames from this test:

. . .

The test does not bother to animate the hero's movement the same way a user's keystrokes would—the hero just jumps. Tests can do that.

Now that the test's current assertions all pass, we can fold more duplication—even before finishing that test's main goal.

```
    class
Game
{
  public:
  //  inanimate objects always construct first;
  //  ego always last
    Spright gun;
    Spright goblin;
```

```
    Spright ego;

Game():
       gun(sprite_gun,           Point(23, 23), 0),
    goblin(sprite_goblin_asleep, Point(70, 60), 1),
       ego(sprite_face,          Point(15, 15), 2)
        {}

    void
paint()
    {
       gun.appear();
    goblin.appear();
       ego.appear();

    vUpdateSpriteMap();
    vFlipScreen(1);
    }
};

    void
test_pickUp()
{
    Game aGame;

  //  jump ego to gun (via cheating):

    aGame.ego.at = aGame.gun.at;
    aGame.ego.pickUp(aGame.gun);

    aGame.paint();

    ASSERT_COLOR( aGame.ego.at.exx + 0,
                  aGame.ego.at.why + 1, 0x1B5 );

  //  jump ego to aim at goblin

    aGame.ego.at.exx = 50;
    aGame.ego.at.why = aGame.goblin.at.why;

    aGame.paint();

    ASSERT_COLOR( aGame.ego.at.exx + 0,
                  aGame.ego.at.why + 1, 0x1B5 );
}
```

We have a new class. What a surprise. Its only method is named after one of our comments. Now push that class into the other two tests, and see if they get smaller:

```
    void
test_moveAsprite()
{
    Game aGame;

    for ( aGame.ego.at = 15;
          aGame.ego.at < 50;
        ++aGame.ego.at )
        {
```

```
        aGame.paint();
        }
    ASSERT_COLOR(49, 49, 0x7C00);
}
```

That one got very short (recall it started at 10 lines of behavior). It previously did not draw a gun or goblin—now it does. The test grows redundant with the other tests. But focus on the `for` loop; it simulates an event message pump. When the time comes to add the real one we may remember this test, and make it useful again. Or maybe not.

The behavior, however, remains incorrect compared to the test that picks up the gun:

```
.....
```

And this test contains the problem:

```
    void
test_pickUpGun()
{
    Game aGame;
    aGame.paint();

    ASSERT_COLOR(23, 23, 0x1B5);

    for ( aGame.ego.at = 16;
          aGame.ego.at < 50;
        ++aGame.ego.at )
        {
        aGame.paint();
        int16_t hit = vSpriteCollision(2, 0, 0);

        if (-1 != hit)
            {
            aGame.gun.at = aGame.ego.at;
            }
        vUpdateSpriteMap();  //  oops
        vFlipScreen(1);
        }
    ASSERT_COLOR(23, 23, 0);
    ASSERT_SAME_POINT(aGame.gun.at, aGame.ego.at - 1);
}
```

Focus on the comment "`oops`". It marks a function call, `vUpdateSpriteMap()`, that now duplicates (logically and physically) the same call inside `paint()`.

I could analyze the situation, and wonder if `paint()` really is the best site for the collision detection system. Recall this must soon bond with the `pickUp()` system. Or, instead of analyzing, I could just blindly fold duplication together and see if all the tests still pass. (But rest assured quite a bit of analysis already occurred. Formerly, that duplication removal was the highest risk option, before removing all the trivial duplication first.)

```
    void
paint()
    {
       gun.appear();
    goblin.appear();
       ego.appear();
```

```
  //  TODO  fix that hard-coded 2 there
    int16_t hit = vSpriteCollision(2, 0, 0);

    if (-1 != hit)
        ego.pickUp(gun);

    vUpdateSpriteMap();
    vFlipScreen(1);
    }
};
```

After moving those lines out of that test and up into `Game::paint()`, both animating tests pick up the gun:

```
....…

...
```

Now all three tests end with the gun aimed at the Goblin.

Bang

We add the definition of an energy bolt from the gun:

```
SPRITE sprite_bolt_east =
{
    0,
VCAPS_RGB332,
     8,  1,
     9,  3,
};

#define cRED c332(7,0,0)
#define cORA c332(7,4,0)
#define cYEL c332(7,7,0)

uint8_t raw_bolt_east[] =  //  the pixel data
{
       0,   0,   0,   0,cRED,cORA,cORA,cYEL,cYEL,
    cRED,cRED,cRED,cORA,cORA,cORA,cYEL,cYEL,c332(7,7,3),
       0,   0,   0,   0,cRED,cORA,cORA,cYEL,cYEL,
};
```

Note the bolt travels to the East, its hotspot (`8, 1`) is on the far right, and this spot is white. The other colors show a hot "flare" trailing the bolt; that helps the illusion of motion.

Then add to the game objects these lines:

```
    class
Spright
{
  public:
…
    void shoot(Game &aGame);
…
};
```

```
        class
    Game
    {
      public:
…
        Spright bolt_east;

    Game():
           gun(sprite_gun,           Point(23, 23), 0),
        goblin(sprite_goblin_asleep, Point(75, 60), 1),
           ego(sprite_face,          Point(15, 15), 2),
    bolt_east(sprite_bolt_east,   Point(69, 58), 3)
            {}
…
    };

        void
    Spright::shoot(Game &aGame)
        {
        aGame.bolt_east.appear();
        }
```

Then tweak the various screen locations to get this:

Finally, after authoring a good bolt, set the tests to ensure the white spot appears 10 pixels to the right of the tip of the gun:

```
        aGame.gun.shoot(aGame);
        aGame.paint();

        ASSERT_COLOR( aGame.ego.at.exx + 9,
                      aGame.ego.at.why - 2, 0x1B5 );

        ASSERT_COLOR( aGame.ego.at.exx + 19,
                      aGame.ego.at.why -  2, 0x7FFF );
```

The programming cycle here isn't strictly test-first. I typically author a visual feature, then add a test based on a pixel color. Test-first won't work yet because the assertions are too hyperactive. I refuse to calculate in my head the offset from ego's hotspot to the tip of the gun, then to the tip of the first bolt.

The alternative is to make the last frame of the last test look like what I want. Then I set the test. Cheating like this is only sustainable within a generally test-first project. The first symptom of this cheating is lower velocity.

The fixes for these bootstrapping issues should be:

- fuzzier assertions that check the general location of items
- test fixtures that address concepts higher level than locations
- tests that address our program's structure instead of GUI details.

We want the next `.paint()` to move the bullet 9 pixels to the right:

```
        aGame.paint();
```

```
        ASSERT_COLOR( aGame.ego.at.exx + 28,
                      aGame.ego.at.why -  2, 0x7FFF );
```

The test fails. The fix adds to each `Spright` a delta—the amount it will move in each frame—and a `m_visible` flag:

```
        class
    Spright
    {
      public:
        SPRITE     &m_aSprite;
        Point       at;
        Point       delta;
        int16_t     m_zee;
        bool        m_visible;
…
    Spright( SPRITE &aSprite, Point at_,  int16_t  zee,
                                                  bool visible ):
            m_pSprite(&aSprite), at(at_), m_zee(zee),
                                              m_visible(visible),
            m_pHolding(NULL)
…
```

Similarly upgrade each `Spright`'s constructor to pass `true`, except `false` for the `bolt_east`.

Change the `shoot()` behavior from calling `paint()` to making the east-pointing `Spright` bolt visible, located just after the gun barrel, and endowed with eastward momentum:

```
        void
    Spright::shoot(Game &aGame)
        {
        aGame.bolt_east.m_visible = true;
        aGame.bolt_east.delta = Point(9, 0);
        aGame.bolt_east.at = Point(at.exx + 10, at.why - 2);
        }
```

Upgrade `paint()` so it's now the only method to call any `appear()`:

```
        void
    paint()
        {
           gun.appear();
        goblin.appear();
           ego.appear();

        if (bolt_east.m_visible)
            bolt_east.appear();
…
```

And upgrade `appear()` to handle `delta`s:

```
        void
    appear()
        {
…
        at.exx += delta.exx;
        at.why += delta.why;
```

```
vSpriteSet(m_zee, m_pSprite, at.exx, at.why);
}
```

Emergently, a test that expects the bolt to appear twice has upgraded the `paint()` system so that bolts, at least, may be invisible, and so any `Spright` has a movement delta.

The last few lines of `test_shootGoblin()` ensure the bolt disappears, and the Goblin wakes up:

```
aGame.paint();

ASSERT_COLOR( aGame.ego.at.exx + 28,
              aGame.ego.at.why -  2, 0x2C0 );

ASSERT_EQUAL( aGame.goblin.m_pSprite,
               &sprite_goblin_awake );
```

Remember when I said that `m_aSprite` should be a reference, not a pointer? We found a reason to turn it back into a pointer. Our `Sprights` now can change their appearance. So I upgrade `SPRITE &m_aSprite` to `SPRITE * m_pSprite`, then change the indirection operators in a few places.

Eventually, bolts should drive a Logic Layer, to provide random damage to affect `Spright`'s health points, so heroes can kill them, and vice versa. For now, this test arbitrarily decrees that shooting a sleeping Goblin at point-blank range only awakens it. To compile the test, clone `sprite_goblin_asleep`, creating `sprite_goblin_awake`, with the only difference that its eyes are open. Then put another collision detector inside `paint()`., and let it turn off the bolt and awaken the Goblin.

```
   void
paint()
    {
    gun.appear();
    ego.appear();

  //  TODO  fix that hard-coded 2 there
    int16_t hit = vSpriteCollision(2, 0, 0);

    if (-1 != hit)
        ego.pickUp(gun);

    hit = vSpriteCollision(3, 1, 1);

    if (-1 != hit)
        {
        goblin.m_pSprite = &sprite_goblin_awake;
        bolt_east.m_visible = false;
        }
    goblin.appear();

    if (bolt_east.m_visible)
        bolt_east.appear();
    else
        vSpriteSet(3, NULL, 0, 0);
```

```
    vUpdateSpriteMap();
    vFlipScreen(1);
    }
};
```

Each feature makes the method `paint()` uglier. This damage indicates some obvious refactors. The `Sprights` must share an array of themselves (paralleling VMPG's internal `SPRITE` slots); each `Spright` must know its own index into the slots, and `appear()` take responsibility for collision detection and state changes.

I like this observation. It places the behaviors that provide value at the top and ensures that the design is the best attainable to support the most important features.

> In traditional development, one designs structures, once, and then debugs until the behavior stabilizes. Here, one writes behavior twice (tests and code), and then refactors until the structure stabilizes.
>
> Unstable behavior is bugs. With wall-to-wall tests, unstable structural design is only a pleasant challenge.

Our next big category of development is user-input. Now that the Goblin is awake, our hero must have the option to run away. To learn how to run a message pump, extend `main()` to run a game after all the tests:

```
    void
zip( uint32_t keys, uint32_t key,
     int16_t &direction, int16_t offset )
{
    if (keys & key && direction < 6 && direction > -6)
        direction += offset;
    else
        if ( (offset > 0 && direction > 0) ||
             (offset < 0 && direction < 0) )
                direction -= offset;
}

    int
main()
{
    test_console();
    test_moveAsprite();
    test_pickUpGun();
    test_shootGoblin();

    Game aGame;
    Point &egoDelta = aGame.ego.delta;

    for(;;)
        {
        uint32_t keys = vGetButtonData();
```

```
            zip(keys, KEY_LEFT , egoDelta.exx, -1);
            zip(keys, KEY_RIGHT, egoDelta.exx, +1);
            zip(keys, KEY_UP   , egoDelta.why, -1);
            zip(keys, KEY_DOWN , egoDelta.why, +1);

            if (keys & KEY_FIRE && aGame.ego.m_pHolding)
                aGame.gun.shoot(aGame);

            aGame.paint();
            }
        return 0;
    }
```

Despite all the missing features and design flaws, this short `main()` provides a playable game! The `zip()` accelerates as a user holds down an arrow key. Releasing a key skids the hero to a halt. Play the game by zipping, then nudging, the hero toward the gun. After picking it up, nudge the gun to take aim at the Goblin, and press the Fire button to wake it up.

Now we must quickly convert that "spike" code into test-first code, before anyone notices that, once again, we wrote it the old-fashioned way.

Mock User

To mock a user, we need to capture the sequence of inputs a user will use to get something done. So temporarily instrument the sample event loop to grab a trace of events, then run the program and copy out the results:

```
…
        for(;;)
            {
            uint32_t keys = vGetButtonData();

            if (keys || egoDelta.exx || egoDelta.why)
                DbgPrintf("%i,",keys);
…
```

The `if` statement checks that `egoDelta.exx` and `egoDelta.why` are not zero. This indicates the Ego is either moving or skidding. `vGetButtonData()` returns `0` each time it's called with no cell phone button pressed.

So the `if` statement censors out most of the unused `0` keys, and logs the ones just after a move. This provides the "skid to a halt" effect. The `DbgPrintf()` provides a trace of keystrokes. Play the game to pick up the gun and wake up the Goblin, then copy the stream of events into a new test:

```
        void
    test_run()
    {
        Game aGame;

        int keyses[] = {
            2,0,2,0,2,2,0,0,8,8,0,0,2,2,2,2,2,2,2,2,
            0,0,0,0,0,0,1,0,1,1,0,0,1,1,0,0,16,16
            };

        for (int x (0);  x < getCount(keyses);  ++x)
            aGame.run(keyses[x]);
```

```
    ASSERT_EQUAL( aGame.goblin.m_pSprite,
                  &sprite_goblin_awake );
}
```

Note, as predicted, we no longer test by sampling pixel colors. The above test checks that those inputs target and awaken the Goblin. The test passes when our `goblin Spright` points to the awakened `SPRITE` object. When low-level tests check pixels to constrain our objects, and high-level ones check these objects, all the tests together constrain development.

To pass the test, move much of our `main()` into a new `Game::run()`:

```
    void
run(uint32_t keys)
    {
    Point &egoDelta = ego.delta;
    zip(keys, KEY_LEFT , egoDelta.exx, -1);
    zip(keys, KEY_RIGHT, egoDelta.exx, +1);
    zip(keys, KEY_UP   , egoDelta.why, -1);
    zip(keys, KEY_DOWN , egoDelta.why, +1);

    if (keys & KEY_FIRE && ego.m_pHolding)
        ego.m_pHolding->shoot(*this);

    paint();
    }
```

When the tests run, the last one now presents an amusing “motion capture” of me grabbing the gun and shooting the Goblin.

And that’s it. The remaining development should aim the gun in different directions, add more critters with more abilities, and put them all in a maze of rooms. A small game like this should reuse and grow the design and test fixtures discussed.

Conclusion

A large game requires the most complex software engineering in the world. Changing high-level content can change play balance. If an artist makes your Goblin darker, you might not notice him in the shadows until it’s too late. Changing content affects gameplay balance; a critical metric that resists automated tests. This Case Study can only scratch the surface of the issues involved.

Our next Case Study investigates another content-rich situation, and it also resists the temptation to just test raw pixels. It leads to the kind of “scenario tests” that highly interactive game design environments require.

Chapter 11: *Fractal Life Engine*

Although fractals and 3D graphics are fun, this Case Study demonstrates a topic more important to all software engineering.

In a complex application, some modules change at different rates, for different reasons, with different risks, and following different paradigms. For example, "Structured Query Language" (SQL) is declarative. Programmers specify a goal, and database engines infer the low-level procedures that supply matching result sets. By contrast, C++ programs are ultimately procedural. Programmers specify statement sequences, and each evaluates, in order, to change data toward the goal. When a complex application requires more than one programming paradigm, its modules often alternate between soft and hard systems.

The teaser:

The left panel is the soft layer; the right is hard

This iteration's tasks:

- Page 307: *Dramatis personæ*
  - 309: Build a frame window
  - 314: Retrofit tests
  - 318: Embed Ruby as a scripting language
- 322: Mock OpenGL Graphics
- 347: Render Flea fractals in OpenGL.

Qt and OpenGL

Previous Case Studies explored the GUI Toolkits Tk and WTL. They are thin OO wrappers on thick underlying GUI platforms—X Windows and MS Windows, respectively. By contrast, Haavard Nord and Eirik Chambe-Eng invented Trolltech's Qt platform entirely in C++. That language's powerful features apply not to wrapping but to architecting Qt, forming a complete application framework and a cross-platform Operating System abstraction layer.

The definitive Qt tutorial is *C++ GUI Programming with Qt 3*, by Jasmin Blanchette & Mark Summerfield. Get Qt from your friendly neighborhood Linux distribution, from `http://www.trolltech.com/`, or from that book's CD.

C++ is very mechanical, with no common garbage collector. Each time you write `new`, creating an object on the heap, you should immediately account for a matching `delete` to free that object, or your program will leak memory. If that object allocated other resources, such as screen area, and if its destructor should return that resource, it might also leak. C++ developers often maintain a tree of object ownership in memory; a trunk object might live in the stack frame of `main()`, while it keeps lists of branch objects, and they maintain lists of leaves. A leaf's constructor takes a pointer to a branch object, and it adds itself to its branch's list. A leaf's destructor removes itself from its branch's list. When `main()` exits, it destroys its trunk

object, and this calls delete for each branch. They call `delete` for each leaf, and all allocated resources are free.

The Composite Pattern maintains this tree of ownership by escalating the ownership behavior into a common base class. Qt calls that `QObject`. Qt maintains a hierarchy of GUI objects—windows, frames, widgets—as a hierarchy in memory. This formalism provides elegant and efficient coding.

To compile this project, I use Visual Studio 7 and Qt's 3.2.1 Non Commercial distribution. All the systems used to assemble this project also work perfectly on Linux, where Qt backs up the mighty "KDE Desktop Environment". Qt is a soft layer written in a hard language.

Silicon Graphics, Inc, developed Open Graphics Library () to render interactive 3D animations on diverse platforms. OpenGL's working model resembles "Assembly Language":

```
glClearColor(0,0,0,1);
glClear(GL_COLOR_BUFFER_BIT);
glBegin(GL_TRIANGLES);        //  begin to construct a Triangle object
           //  X     Y     Z
  glVertex3d(-1.0, 0.0, 0.0);  //  constructor parameters
  glVertex3d( 0.0, 1.0, 0.0);
  glVertex3d( 1.0, 0.0, 0.0);
glEnd();                      //  push the Triangle into the current display context
```

Zillions of such primitive function calls assemble OpenGL images. Each call accumulates small data types—bits, floats, etc.—into hidden objects. Similarly, assembly language performs countless small operations against relatively invisible hardware components. OpenGL naturally tunes to drive 3D graphics hardware; drivers can easily translate its display lists into their specific hardware operations.

Note how OpenGL programmers format those lines. To symbolize the behavior of the invisible Triangle object, behind those functions, they indent the `glVertex3d()` calls that provide the Triangle's three corners. These styles fulfill Larry Wall's infamous prophecy, "Good programmers can write assembler in any language."

OpenGL is a very hard layer, so we need a very soft layer to drive it.

Flea (and POVray)

Fractal Life Engine (Flea), my first Ruby project, produces art like this:

Flea uses "Lindenmayer Systems" to design fractals, per the influential book *The Algorithmic Beauty of Plants* by Przemyslaw Prusinkiewicz and Aristid Lindenmayer. To draw a tree, an LSystem declares the behavior of an invisible turtle. Imagine a turtle crawling up that tree's trunk. At the first cluster of limbs, the turtle splits into six smaller turtles. Five crawl up the lateral branches, and one crawls up the (shaded) stem. At each branch, a turtle splits into smaller turtles. When the turtles grow small enough, they crawl around the perimeter of those green leaves.

To specify a shape with turtle graphics, place a turtle on the ground, point it upright, and give it a list of instructions. "Crawl for 5 of your body lengths, drawing tree bark as you go. Then get smaller, rotate to the left, split into two turtles, change your angle, repeat these instructions, and when you get small enough, draw a leaf and retire." The turtle obeys the instructions, and passes them to its spawned babies. They draw a recursive shape as they go.

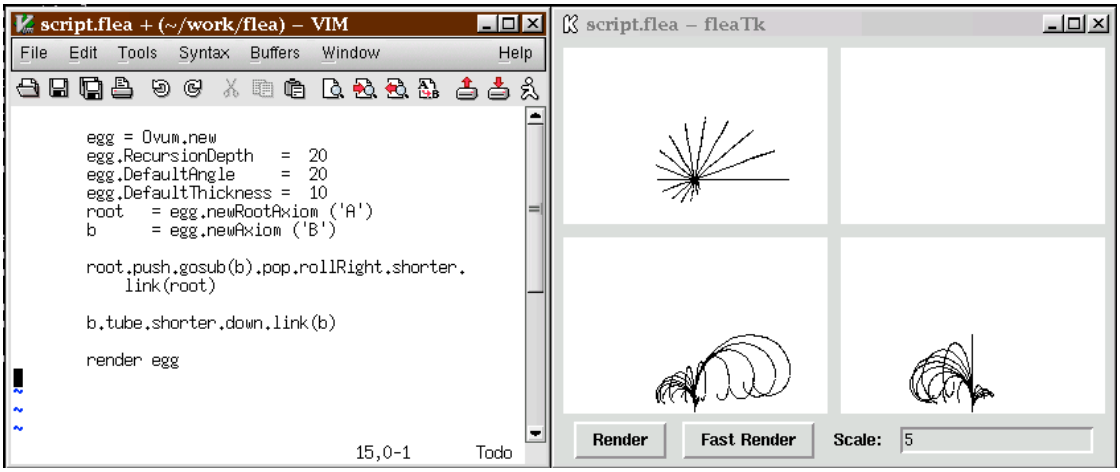
In theory, this system defines real fractals, as turtles spawn smaller turtles, *ad-infinitum*. In practice, with finite CPU time, turtles get so small they become irrelevant, and we stop

worrying about them. This Case Study will casually refer to Flea output as “fractals”, though not all Flea scripts produce self-similarity.

Instructing a turtle to get bigger, spawn more turtles, and repeat is not a good idea.

Flea allows users to program turtles with sequences of commands that draw shapes like trees. Then Flea operates these turtles, and records the thousands of low-level commands they emit. Flea feeds the low-level commands into one of two kinds of pluggable backends—sketchers and renderers.

Users author Flea scripts using a sketching backend:



The window on the left is the Vim editor, and on the right is the `fleaTk` sketching backend. Users edit a Flea script in any editor, and each time they save the file, `fleaTk` detects the file changed, and repaints the results. `fleaTk` displays fractals in three flat views, each expressing two of the three possible dimensions. (The upper left panel could upgrade to show a perspective sketch.)

When a fractal looks good, users then send the same script to a 3D rendering system, such as “Open Inventor”, “Virtual Reality Modeling Language”, or POVray.

We could write a high-quality rendering backend with OpenGL. To close our loop as soon as possible, this Case Study will only start a minimal sketching backend. We will render Flea outputs using only the most primitive OpenGL features, without shading or colors. To exceed `fleaTk`’s abilities, we will add an animated rotation feature.

To get Flea working on your system, download it from `http://flea.sourceforge.net/`, and install Ruby and POVray. This project won’t need POVray, but a second opinion is always good. The *SVG Canvas* Case Study, for example, used `dot`’s PNG output to double-check its canvas. Similarly, this project could use Flea’s POVray output to double-check our output. Get the most amazing ray-tracing system, POVray, from `http://www.povray.org/`, and try not to lose yourself in its vivid universe.

If Flea interests you more than Qt or OpenGL, activate that `fleaTk` interface, and skip to page 357 to see how to grow fractals with it.

Main Window

Qt’s architectural elegance leads to easy testing, and we could start this project with a test. However, the *NanoCppUnit* Case Study, on page 167, has already shown how to bootstrap a difficult GUI with a simple test (and provided some introductory C++ tips). This Case Study

will develop our first window using Code-and-Fix, then bootstrap test-first. That leads to a different technique.

TFP purists need not fret. After creating a Qt window, the next big feature requires major research to achieve pure test-first. Qt is just too easy! This book is about hard situations.

Sometimes a GUI platform presents a very pernicious environment that cannot efficiently decouple. Either too much programmer time or processor time will be wasted enforcing Test Isolation. Sometimes each test case cannot build a top-level frame window, test it, and destroy it. Maybe, for example, you are commissioned to invest only a small effort into a legacy GUI, lacking tests, and can't risk decoupling the feature you need from the features it needs.

In these situations, you might find yourself using *in-vivo* testing (from *Working Effectively with Legacy Code* by Mike Feathers). Allow your program's usual `main()` to build your application's main window, and leave it up. Then insert a call to a test rig, and ensure each case drives this main window, like a user, and returns it to a rest state between tests. This situation risks losing some Test Isolation benefits. Strict Test Isolation destroys and rebuilds everything, on principle. A hard situation should compromise with assertions that check each test case returns an application to a default state.

Now pretend to forget to write tests, and build a main window. Install Qt3, launch Visual Studio 7, and use New Project → Visual C++ Projects → Win32 → Win32 Console to create a new program called FleaGL. The Visual Studio Wizard will produce this:

```
#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
    return 0;
}
```

The first line, `#include "stdafx.h"`, triggers one of the VC++ precompiled header systems. All projects using this system use two anchor files, typically called "`stdafx.h`" and "`stdafx.cpp`". VC++ compiles them files into a binary symbol table. To compile any other file that `#includes "stdafx.h"`, VC++ re-reads that binary symbol table. This system avoids re-compiling tens of thousands of lines of header files after each change. Projects should put `#include` directives for any header files they don't plan to edit into "`stdafx.h`". After I introduce our project's Qt and C++ Standard Library headers, I will migrate them into it without comment.

Because all modules in a project generally begin with `#include "stdafx.h"`, projects should *not* put things into that file that not everyone should know, or things that change frequently.

(Irritatingly, VC++ cannot compile lines appearing above `#include "stdafx.h"`. This breaks the style guideline, "The first line of '`foo.cpp`' shall be `#include "foo.h"`." That rule tests that header files `#include` all their dependencies, with no surprises.)

To tell our new project where Qt is, ensure you have installed Qt properly (and possibly rebooted), and that your `QTDIR` environmental variable is valid. Then use Project → Properties → C/C++ → General → Additional Include Directories, and add "`$(QTDIR)\include`". Switch to Linker → General → Additional Library Directories, and add "`$(QTDIR)\lib`". Switch to Linker → Input, and add "`qt-mtnc321.lib qtmain.lib`".

Now upgrade our empty program into a minimal Qt application with a main window:

```
#include "stdafx.h"
#include <qapplication.h>
```

```
#include <qmainwindow.h>

int _tmain(int argc, _TCHAR* argv[])
{
    QApplication app( argc, argv );
    QMainWindow aMainWindow;
    app.setMainWidget(&aMainWindow);  //  give the application a root widget
    aMainWindow.setGeometry(10,20,800,450);  //  get large

    app.connect  //  Observer Pattern
            (
            &app, SIGNAL( lastWindowClosed() ),
            &app, SLOT( quit() )
            );

    aMainWindow.show();  //  paint on the screen
    return app.exec();  //  "mainloop()"
}
```

Run that, and predict a big grey window with nothing in it.

The call to `.connect()` does more than allow closing that window to terminate our application. `.connect()` is the heart of Qt programming. When anything calls `app.lastWindowClosed()`, the "Signals and Slots" system calls `app.quit()` automatically.

Qt's architects observed that the Observer Pattern was so important to programming in general, and GUIs in particular, they built the pattern directly into C++. They wrote a pre-compiler called `moc` to add to C++ the special keywords `signals` and `slots`. (Our code has not used them yet; in a couple pages we must add `moc` to our compilation system.) The class definition for `QApplication` contains these highlights:

```
class Q_EXPORT QApplication : public QObject
{
    Q_OBJECT

public:
…
signals:
    void         lastWindowClosed();
    void         aboutToQuit();
    void         guiThreadAwake();

public slots:
    void         quit();
    void         closeAllWindows();
    void         aboutQt();
…
};
```

A program can configure any object's signal to send a notification to any other object's slot. The objects themselves remain decoupled; they don't need to know anything about each other. Their containing object knows about both of them and uses `.connect()` to link them.

Now we add to our window two edit fields, and a widget to render OpenGL. We divide these with splitter bars. To give these objects a home, we derive a new main window class from `QMainWindow`, and put it into a new file, "`FleaGL.h`":

```
#include <qmainwindow.h>
```

```
class QTextEdit;  //   forward declaration to prevent long compiles
class QGLWidget;
class QSplitter;

    class
MainFleaWindow: public QMainWindow
{
    Q_OBJECT  //   register with moc system

public:
    MainFleaWindow();
    void pullTheMiddleSplitToTheLeftALittle();

    QTextEdit *sourceEditor;
    QTextEdit * traceEditor;
    QSplitter *midSplit;
    QGLWidget *glWidget;

};
```

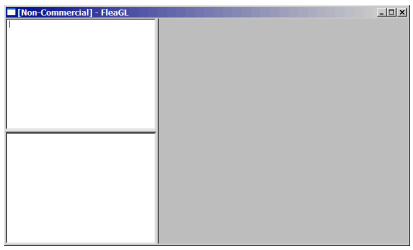
Now add these to "FleaGL.cpp". Notice, while constructing a MainFleaWindow, that each widget constructor takes a pointer to its parent widget:

```
#include "stdafx.h"
#include "FleaGL.h"  //  include first to prove it doesn't need these:
#include <qtextedit.h>
#include <qsplitter.h>
#include <qgl.h>

MainFleaWindow::MainFleaWindow() :
            sourceEditor(NULL),
             traceEditor(NULL)
{

  //  the primary splitter divides left from right [ | ]

    midSplit = new QSplitter(Horizontal, this);

  //  a secondary splitter divides upper left from lower left [-| ]

    QSplitter *leftSplit = new QSplitter(Vertical, midSplit);

  //  the right panel is the OpenGL widget

    glWidget = new QGLWidget(midSplit);

  //  the upper and lower left widgets are text editors

    sourceEditor = new QTextEdit(leftSplit);
     traceEditor = new QTextEdit(leftSplit);

    pullTheMiddleSplitToTheLeftALittle();

  //  plug the root object into the frame

    setCentralWidget(midSplit);
```

```
        midSplit->show();  //  light their pixels up
    }

        void
    MainFleaWindow::pullTheMiddleSplitToTheLeftALittle()
    {
        QValueList<int> sizes;
        sizes.append(300);
        sizes.append(500);
        midSplit->setSizes(sizes);
    }

    int _tmain(int argc, _TCHAR* argv[])
    {
        QApplication app( argc, argv );
        MainFleaWindow aMainWindow;
…
    }
```

That owner object constructs its child objects directly in its constructor. Because Qt is not a wrapper, `new` creates real widget objects, themselves, directly. This simplifies our mental effort. To construct the child objects of `MainFleaWindow`, we need no `WM_INITDIALOG` message, or `OnInitDialog()` handler, to construct children only when the GUI Toolkit announces it is ready. And when our window destroys, its base class destructors call `delete` on all child controls. Our derived class requires no explicit destructor.

However, the code can't compile and link yet. `MainFleaWindow` uses `Q_OBJECT`, to prepare for `signals` and `slots`. If you comment `Q_OBJECT` out, you can compile and run. That constructor produces a window with two splitter bars, two editors, and a mysterious grey area:



Before going further, we must add the keywords `signals` and `slots` to VC++.

Meta Object Compiler

Qt's `moc` system provides the signals and slots mechanism for inter-object communication, runtime type information, and the dynamic property system. To plug it into a Visual Studio 7 application, install the QMsNet toolbar onto your editor, open "`FleaGL.h`", and click on the toolbar button with a tooltip saying, "This will add a MOC step for the current file."

That button adds a "Custom Build Step" that generates "`tmp\moc\moc_FleaGL.cpp`". `Q_OBJECT` declared extra `MainFleaWindow` methods, and the new `.cpp` file implements them, to back up the signals and slots system.

To Trolltech's credit, adding two new keywords to many different C++ compilers, on many different platforms, requires surprisingly few extra configurations. But they could not think of everything. If you follow my exact instructions (instead of whatever the Qt documentation says), you'll get this error message:

```
…\tmp\moc\moc_FleaGL.cpp(93) : fatal error C1010: unexpected end of file
                    while looking for precompiled header directive
```

That's because "moc_FleaGL.cpp" did not #include "stdafx.h", which is a good thing. Fix VC++ by selecting Solution Explorer → Generated MOC Files → moc_FleaGL.cpp → Context Menu → Properties → C/C++ → Precompiled Headers → Create/Use Precompiled Header → Not Using Precompiled Headers.

Now compile and execute the program; it should look like that screen shot.

But we "forgot" to test first!

In-Vivo *Testing*

Let's pretend our simple main() function were instead elaborate, complex, crufty, or written using an inferior GUI Toolkit. In some situations, when retrofitting tests into legacy code, the risk of breaking up a complex main() function exceeds the benefit of writing a few tests. When you have legacy code, and a bug report, rapidly writing a test to capture that bug should be your highest priority. The purity of your Test Isolation should be secondary.

To retrofit tests, we need a test rig. You could get CppUnit, or any of its clones, online. For continuity, we will extract the test rig begun in the *NanoCppUnit* Case Study. It finished upgrading during the *Broadband Feedback* Case Study. Go to its source code, open the file "Project_test.cpp", and copy everything from #include <sstream> to #define RETEST_(). Go to FleaGL's folder, create a new file called "test.h", and paste all that test stuff in.

Take out the TestDialog class definition (I put it out of order!), and search and replace some CStringA types and replace them with std::string.

Now trim the file's head and tail a little, like this:

```
// a NanoCppUnit in one header...

#ifndef TEST_
#   include <list>
#   include <iostream>
#   include <sstream>
// #   include <atlCtrls.h>
…
// bool TestCase::all_tests_passed = true;
// TestCase::TestCases_t TestCase::cases;
…
#endif  //  an atypical C++ "Include Guard"
```

Also comment-out those class static members. Remember—code never truly decouples until you re-use it!

Bond the new "test.h" with the FleaGL project's Header Files list, and #include it into "FleaGL.cpp":

```
#ifdef _DEBUG
#   include "test.h"

    bool TestCase::all_tests_passed = true;
    TestCase::TestCases_t TestCase::cases;

TEST_(TestCase, testTheTests)
{
    CPPUNIT_ASSERT_EQUAL(true, true);
}
#endif
```

Because `_DEBUG` mode has extra code for assertions, and it activates debugger support, we will conditionally compile all test code between that `#ifdef _DEBUG` and `#endif`.

Recompile and test. You will get the same window as before. The tests did not run.

Create a trivial test runner, and edit `main()` to conditionally call the tests:

```
    #ifdef _DEBUG
…
        bool
    runTests()
    {
        bool result = TestCase::runTests();

        char const * disposition = "Tests failed...\n";
        if (result)  disposition = "All tests passed!\n";
        cout << disposition << std::flush;
        OutputDebugStringA(disposition);
        return result;
    }
…
    #endif
…
    int _tmain(int argc, _TCHAR* argv[])
    {
        QApplication app( argc, argv );
        MainFleaWindow aMainWindow;
        app.setMainWidget(&aMainWindow);
        aMainWindow.setGeometry(10,20,800,450);

        app.connect
                (
                &app, SIGNAL( lastWindowClosed() ),
                &app, SLOT( quit() )
                );

        #ifdef _DEBUG
            return ! runTests();
        #else
            aMainWindow.show();
            return app.exec();
        #endif
    }
```

Find `CPPUNIT_ASSERT_EQUAL(true, true)`, put a breakpoint on it, run the tests, and prove control flow gets there. The VC++ Output panel will then display "`All tests passed!`" among its usual reports.

Our last little task is a Learner Test. We will merely Set and Get a string into the upper left editor panel. Our test plan, in whiteboard C++, is this:

```
EST_(TestCase, testSetGet)

QString sample = "puts('hello world')\n";
aMainWindow.sourceEditor . . . insert text( sample );
QString result = aMainWindow.sourceEditor . . . GetText();
CPPUNIT_ASSERT_EQUAL(result, sample);
```

(Our imaginary team has not yet learned to insert text into a `QTextEdit` with `.setText()`, or to extract it with `.text()`.)

The first problem will be the `aMainWindow` object. It lives down in `main()`, and we pretend we can't take it out. We built the current design without tests, so it doesn't lead freely to easy testing. This will demonstrate *in-vivo* testing for GUIs.

Find the case with `CPPUNIT_ASSERT_EQUAL(true, true)`, and replace it with that plan. Then get it to pass by any means necessary:

```
    #include "stdafx.h"
    #include "FleaGL.h"

        static MainFleaWindow *pMainWindow;
        static QApplication   *pApp;  //  reveal() will need this
…

    TEST_(TestCase, testSetGet)
    {
        QString sample = "puts('hello world')\n";
        pMainWindow->sourceEditor->setText(sample);
        QString result = pMainWindow->sourceEditor->text();
        CPPUNIT_ASSERT_EQUAL(result, sample);
    }
…
    int _tmain(int argc, _TCHAR* argv[])
    {
        QApplication app( argc, argv );
        MainFleaWindow aMainWindow;

        pMainWindow = &aMainWindow;
        pApp        = &app;

        app.setMainWidget(&aMainWindow);
…
    }
```

If our `main()` had some flaw that actually prevented Test Isolation, pointing each test case to the same single instance of our main window is an acceptable compromise, so long as each test case leaves the window ready for the next test. In a flawed situation, getting *any* tests working must come before refactoring to decouple things.

Our test cases will access the main window using a static pointer. They could have used different techniques. `main()` could pass a reference to the main window into the test runner, and

this would pass the reference to each test. That strategy would complicate the test rig, and hypothetically interfere with other projects that share the test rig. Our test rig should decouple from its test targets, and the `static` keyword is as good a C++ decoupling mechanism as more complex systems.

Regulate the Qt Event Queue

I almost forgot. The TFUI Principles, back on page whatever, insist that every new project proactively write a `reveal()` method, to permit rapid review of test cases' target windows in their tested state. In this situation, such a method will help us manually check that our patches on Test Isolation are not causing friction. Later, this fixture helps aggressively test OpenGL:

```
    struct
TestQt: virtual TestCase
{
    void
reveal()
    {
    pMainWindow->show();
    pApp->exec();
    }

    void
setSource(QString const & source)
    {
    pMainWindow->sourceEditor->setText( source );
    }
};

TEST_(TestQt, testGetSet)
{
    QString sample = "puts('hello world')\n";
    setSource(sample);
    QString result = pMainWindow->sourceEditor->text();
    CPPUNIT_ASSERT_EQUAL(result, sample);
    reveal();
}
```

Comment that `reveal()` out when you finish *Temporarily Visually Inspecting* the result:

Now we turn our attention to that mysterious "hello world" string. It appeared on our imaginary whiteboard, and traveled through our functioning test into that upper left edit panel. So where is it going?

Embed a Language

The whiteboard version of our next test is:

```
TEST_(TestQt, HelloWorld)
{
   setSource( "puts('Hello World')\n" );
   pMainWindow->evaluate();  //  write this!
   QString result = pMainWindow->traceEditor->text();
   CPPUNIT_ASSERT_EQUAL(result, "Hello World\n");
}
```

To implement that new method, `MainFleaWindow::evaluate()`, we must evaluate the command `puts()`, inside that input string. We could invent yet another language, and grow it alongside this application, but inventing new languages is a terrible habit. Many fine scripting languages already exist that can easily embed inside a larger program.

Can you guess which language I'll pick?

```
#pragma warning(disable: 4312)  //  turn off a minor warning

#include "C:/ruby/lib/ruby/1.8/i386-mswin32/ruby.h"

  //  this tells the linker to get Ruby's lib file
#pragma comment(lib, "C:/ruby/lib/msvcrt-ruby18.lib")

#undef connect
#undef close
#undef accept  //  these must be above <qapplication.h>

#include <qapplication.h>
...
    class
MainFleaWindow: public QMainWindow
{
    Q_OBJECT

public:
    MainFleaWindow();
    void evaluate();

    QTextEdit *sourceEditor;
    QTextEdit * traceEditor;
    QGLWidget *glWidget;
};
...
    void
MainFleaWindow::evaluate()
{
    QString source = sourceEditor->text();

    ruby_init();
```

```
    ruby_script("embedded");
    rb_eval_string(source);

}
```

That's the minimum implementation of `.evaluate()`. Although it passes a smoke test, it's not enough for our needs. Complications come from two sources. Ruby might throw its version of an exception, which we must translate into an error message. And that test will want the Ruby result to appear in the lower editor. It currently floats up into the "Great Bit Bucket in the Sky".

To fix the first problem, we trap errors with `rb_eval_string_protect()`. That sets `state` to a non-`0` number if an error occurs. Then we convert Ruby's error variable, `$!`, into a string with "`$!.to_s()`". That produces a human-readable error message:

```
    void
MainFleaWindow::evaluate()
{
    QString source = sourceEditor->text();
    traceEditor->setText("");

    ruby_init();
    ruby_script("embedded");

    int state = 0;
    rb_eval_string_protect(source, &state);

    if (state)
        {
        VALUE c = rb_funcall(rb_gv_get("$!"), rb_intern("to_s"), 0);
        traceEditor->insert("ERROR ");
        traceEditor->insert(RSTRING(c)->ptr);
        traceEditor->insert("\n");
        }
}
```

That does not crash, so this much of our test passes:

```
TEST_(TestQt, HelloWorld)
{
    setSource( "puts('Hello World')\n" );
    pMainWindow->evaluate();
    QString result = pMainWindow->traceEditor->text();
//    CPPUNIT_ASSERT_EQUAL(result, "Hello World\n");
    reveal();
}
```

Experiment (via *Temporary Interactive Tests*) by replacing the input with bad Ruby syntax, and read their error messages.

Now that error messages go into our output panel, we need ordinary output to appear there too. Ruby permits redirecting output from `puts()`, `print()`, etc. by creating a new class and assigning `$stdout` to an instance of it:

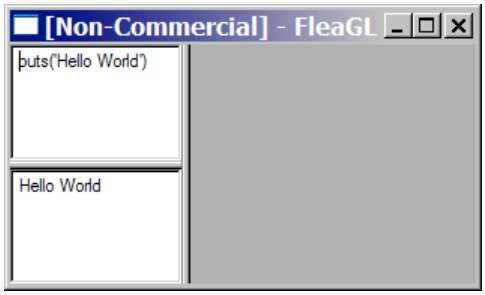
```
class MockStream
    def write(x)
        sink(x)  #   we must write this, in C++, next
    end
end
```

```
$stdout = MockStream.new()
```

Ruby will compile that code even if `sink()` does not exist yet. Only calling `puts()` will call `sink()`, and trigger an error message. De-comment the assertion in that test case, and add these lines to make it pass:

```
    static VALUE
sink(VALUE self, VALUE input)
{
    char * got = StringValueCStr(input);
    pMainWindow->traceEditor->insert(got);
    return Qnil;
}

    void
MainFleaWindow::evaluate()
{
    QString source = sourceEditor->text();
    traceEditor->setText("");

    ruby_init();
    ruby_script("embedded");
    rb_define_global_function("sink", RUBY_METHOD_FUNC(sink), 1);

    source = "class MockStream\n"
             "    def write(x)\n"
             "        sink(x)\n"
             "    end\n"
             "end\n"
             "$stdout = MockStream.new()\n"
             + source;

    int state = 0;
    rb_eval_string_protect(source, &state);

    if (state)
        {
        VALUE c = rb_funcall(rb_gv_get("$!"), rb_intern("to_s"), 0);
        traceEditor->insert("ERROR ");
        traceEditor->insert(RSTRING(c)->ptr);
        traceEditor->insert("\n");
        }
}
```

Registering the C++ function `sink()` with Ruby permits our primitive editor to fulfill its "`Hello World`" mission:



Turning our attention to this mysterious grey area on the right, what combination of C++ and Ruby can fill it up with 3D fractals?

Flea does OpenGL

Flea's input system builds lists of turtle graphics commands as objects of classes derived from `Action`. Then it positions a turtle in 3D space, and links it to an object derived from the class `Backend`. Flea calls each `Action` in the turtle's list to update the turtle's relative position and orientation. When a turtle executes a `tube` command, it calls `Backend::addCylinder()`. Similarly, the turtle's `sphere` command calls `Backend::addSphere()`.

We scribble our goal on our whiteboard of the imagination—a top-level test like this:

```
TEST_(TestQt, Tee)
{
  // from the tube entry on
  // http://flea.sourceforge.net/reference.html

  setSource( "egg = startOvum(1, 90, 100)
              tubes = egg.newRootAxiom('A')
              tubes.longer(5).
                  tube.tube.shorter.
                  push.right.tube.pop.
                  left.tube
              render(egg)" );  // draws a T

  pMainWindow->evaluate();
  ?? shapes = pMainWindow->openGL->getShapes();
  ASSERT(shapes[0] is a vertical tube);
  ASSERT(shapes[1] is a higher vertical tube);
  ASSERT(shapes[2] is a shorter horizontal tube);
  ASSERT(shapes[3] is a shorter horizontal tube);
}
```

That plan generates many intermediate goals. Previous Case Studies, using higher-level systems, would have simply written a failing test, then written some Child Tests, then passed them all. This situation requires a little more research. We don't write that test yet; we leave it on the whiteboard for now.

Before writing a test that operates a Flea script, we need low-level tests that demonstrate OpenGL displaying a tube and a sphere.

Primitive systems like OpenGL are test-hostile, so we must start at an even lower level. To research testing OpenGL, examine a little sample of it:

```
glBegin(GL_TRIANGLE_STRIP);
  for (i = 0; i <= numEdges; i++)
  {
      glNormal3f(x[i], y[i], 0.0);
      glVertex3f(x[i], y[i], bottom);
      glVertex3f(x[i], y[i], top);
  }
```

```
glEnd();
```

Those coordinates go either directly into hardware, or into a display list. OpenGL supports no complete mechanism to query a hardware display list, once written. Further function calls can only modify them and send them into hardware. So much of OpenGL graphics are non-retained.

If those functions were methods, with an arrow like "`thing->glBegin()`", we could use the Mock Graphics pattern to allow tests to re-point `thing` to an object that logged every call to a Log String. Then tests could sample that string for evidence of cylinders and spheres.

OpenGL has no `thing` object to mock. But we must retain its non-retained graphics, to test them.

Mock a DLL

The trick is recognizing that dynamic link libraries already provide `thing->glBegin()`. When MS Windows loads your program, the loader finds every DLL it needs, and redirects all the function calls into it. If we make this process explicit, we can redirect it.

One caveat: OpenGL has a few systems to *Query Visual Appearance*—most notably `glFeedbackBuffer()`. Complex OpenGL projects could use it, with complex test fixtures. This Case Study demonstrates a brute-force technique that works on any dynamic link library. After our DLL submits to tests, we will analyze these retained graphics using the same general patterns as tests that used alternate OpenGL output systems.

To research our next step, I played around for a while. I copied the source code into a folder called `scratch`, and copied in some OpenGL sample code to draw primitives. I wrote scrappy little `TEST_()` cases that only worked as platforms for debugging and visual inspections, to help me learn how these libraries work. Then I wrote the beginning of a Mock Graphics system. All that new code is a "Spike Solution"; it wouldn't go into a real codebase. To bootstrap testing for OpenGL, I use that new code as a "Cheat Sheet", and copy things that work into the real project. Slowly writing crufty code in a scratch project is a very useful technique that leads to rapidly writing elegant code in a production codebase.

The first thing to copy from the exploratory testing into our test code is a class, `OpenGLWrapper`. It will soon provide Mock Graphics. To illustrate how it works, I will pretend to grow it organically here. Don't type all these steps in.

Operating systems load some libraries dynamically. When a program loader launches an executable, it reads secret data in its file to learn the names of each library module it needs, the names of each function, and "Fixup" information. The loader plugs all the functions into the program's stub locations, converting them into real calls to real functions.

Our mission is to replace automatic dynamic linking with programmatic dynamic linking. That gives us a way to spoof each function and replace it with a function that writes to a Log String when production code calls it. Test cases will inspect the Log String for the correct features.

Under MS Windows, a program registers a DLL, "`Foo.dll`", by statically linking with a stub library, "`Foo.lib`". When such a program runs on a computer with no "`Foo.dll`" in any valid folder, the loader aborts and displays a familiar error message.

Programs also dynamically link to libraries by calling the same functions as the loader calls. MS Windows uses `LoadLibrary()`, and most other platforms use `dlopen()`. The following concept will work, with a few type adjustments, on any platform that supports dynamic link libraries:

```
HMODULE libHandle = LoadLibrary("opengl32.dll");
```

```
    void (APIENTRY *glBegin) (GLenum mode);
    FARPROC farPoint = GetProcAddress(libHandle, "glBegin");
    assert(farPoint);
    glBegin = reinterpret_cast<something *> (farPoint);
//  use glBegin() like a function
    FreeLibrary(libHandle);
```

The first line finds OpenGL's DLL and returns its handle. The next line defines a function pointer. OpenGL's core header file, typically called "`GL/gl.h`", defines `glBegin()` like this:

```
WINGDIAPI void APIENTRY glBegin (GLenum mode);
```

Our little code sample declares a very similar local variable, with the same name:

```
    void (APIENTRY *glBegin) (GLenum mode);
```

That's a pointer to a function. `WINGDIAPI` went away, because the pointer itself lives in this module, not the Windows GDI module. `APIENTRY` went inside the parentheses, to modify the function's call type. To read a complex C++ type declaration, say the innermost name, and say the name of each modifier around it, in order of precedence: "`glBegin` points to an `APIENTRY` function that takes a `GLenum` and returns `void`."

On the next line, `GetProcAddress()` reaches inside the OpenGL DLL, and grabs a function called "`glBegin`". But it returns a `FARPROC`, which is not a pointer to an `APIENTRY` function that takes a `GLenum` … etc.

The line with `reinterpret_cast<>()` converts that arbitrary function pointer into 32 raw bits, and then reinterprets these as a pointer to the real `glBegin()`. I wrote *`something`* to hold the place of a complex and high-risk type declaration. Here's the complete version of that line:

```
    glBegin = reinterpret_cast
                  <
                  void (APIENTRY *) (GLenum mode)
                  > (farPoint);
```

My C++ formatting styles often look … different. They give readers a remote chance to comprehend statements like that one. A better strategy avoids such statements.

To reduce the risk, and avoid writing the complete type of `glBegin()` and every other function we must spoof more than once, we upgrade the situation to use a `template`:

```
    template<class funk>
    void
get(funk *& pointer, char const * name)
{
    FARPROC farPoint = GetProcAddress(libHandle, name);
    assert(farPoint);
    pointer = reinterpret_cast<funk *> (farPoint);
}  //  return the pointer by reference
```

That template resolves the high-risk *`something`* for us. It collects `glBegin`'s target's type automatically, puts it into `funk`, and reinterprets the return value of `GetProcAddress()` into `funk`'s pointer's type. Pushing the risk down into a relatively typesafe method permits a very clean list of constructions for all our required function pointers:

```
    void (APIENTRY *glBegin) (GLenum mode);
```

```
    void (APIENTRY *glEnd) (void);
    void (APIENTRY *glVertex3f) (GLfloat x, GLfloat y, GLfloat z);

    get(glBegin    , "glBegin"    );
    get(glEnd      , "glEnd"      );
    get(glVertex3f , "glVertex3f" );
```

That strategy expresses each function pointer's type—the high risk part—once and only once. (And note how the `template` keyword helps C++ static type system behave a little like a dynamic language.) The type is high risk because if we get it wrong, `reinterpret_cast<>` leads to garbage opcodes, undefined behavior, civil unions between dogs and cats, and so on.

That strategy also formats our remaining duplication into tables, sorted for easy scanning. To add a new function:

- copy the function's prototype from "`GL/gl.h`" to the pointer list,
- convert the prototype into a function pointer,
- add the pointer and function name to the `get()` list, and
- upgrade the production code to use the pointer instead of the "real thing".

Production code will use this system as a very thin layer on top of the real OpenGL. We must put all that stuff into a class, and declare a global object of that class, so production code can use it. So here's the class:

```
    #include <assert.h>
…
        class
    OpenGLWrapper
    {
      public:
        OpenGLWrapper(char const * lib = "opengl32.dll"):
            libHandle(LoadLibrary(lib))
        {
            assert(libHandle);
            get(wglCreateContext, "wglCreateContext");
            get(glBegin       , "glBegin"        );
            get(glClear       , "glClear"        );
            get(glEnable      , "glEnable"       );
            get(glEnd         , "glEnd"          );
            get(glFrustum     , "glFrustum"      );
            get(glLineWidth   , "glLineWidth"    );
            get(glLoadIdentity, "glLoadIdentity" );
            get(glMatrixMode  , "glMatrixMode"   );
            get(glNormal3d    , "glNormal3d"     );
            get(glPopMatrix   , "glPopMatrix"    );
            get(glPushMatrix  , "glPushMatrix"   );
            get(glRotated     , "glRotated"      );
            get(glScaled      , "glScaled"       );
            get(glTranslated  , "glTranslated"   );
            get(glVertex3d    , "glVertex3d"     );
            get(glVertex3dv   , "glVertex3dv"    );
        }

        HGLRC (APIENTRY *wglCreateContext) (HDC a);
        void (APIENTRY *glBegin) (GLenum mode);
        void (APIENTRY *glClear) (GLbitfield mask);
        void (APIENTRY *glEnable) (GLenum cap);
        void (APIENTRY *glEnd) (void);
```

```
    void (APIENTRY *glFrustum) (GLdouble left, GLdouble right, GLdouble bottom,
                  GLdouble top, GLdouble zNear, GLdouble zFar);
    void (APIENTRY *glLineWidth) (GLfloat width);
    void (APIENTRY *glLoadIdentity) (void);
    void (APIENTRY *glMatrixMode) (GLenum mode);
    void (APIENTRY *glNormal3d) (GLdouble nx, GLdouble ny, GLdouble nz);
    void (APIENTRY *glPopMatrix) (void);
    void (APIENTRY *glPushMatrix) (void);
    void (APIENTRY *glRotated) (GLdouble angle, GLdouble x, GLdouble y,
                                                              GLdouble z);
    void (APIENTRY *glScaled) (GLdouble x, GLdouble y, GLdouble z);
    void (APIENTRY *glTranslated) (GLdouble x, GLdouble y, GLdouble z);
    void (APIENTRY *glVertex3d) (GLdouble x, GLdouble y, GLdouble z);
    void (APIENTRY *glVertex3dv) (const GLdouble *v);

   ~OpenGLWrapper()
    {
        BOOL freed = FreeLibrary(libHandle);
        assert(freed);
    }

  private:

    template<class funk>
    void
get(funk *& pointer, char const * name)
    {
    FARPROC farPoint = GetProcAddress(libHandle, name);
    pointer = reinterpret_cast<funk *> (farPoint);
    assert(pointer);
    }

    HMODULE libHandle;
};

static OpenGLWrapper ogl;
```

Note the object `ogl`. When it lives outside any function, it constructs before `main()` starts and destructs after `main()` returns, paralleling the real OpenGL DLL. Our production objects will use a pointer that usually points to `ogl`, and our tests will construct a Mock Object and re-point production objects to it.

(In terms of pointer-safety, that object has many pointers, and its constructor does not set any of them to NULL. We remain safe so long as the only instance of this object is global or static. Such objects are set)

OpenGL programmers stress over every CPU cycle. This book, targeting slow GUIs that respond in user time, has avoided that stress, until this Case Study. Interactive animation should avoid even the slight overhead of dereferencing indirect calls like `thing->glBegin(…)`. A real test-first OpenGL project should conditionally compile, say, "`GL_ glBegin(…)`," where `GL_` becomes `thing->` at test build time, and nothing at production build time.

Sane OpenGL Subset

You probably think none of these techniques qualify as sane. Projects need subsets when their design space could be arbitrarily huge. OpenGL's core API provides 420 functions. Like assembly language, most of them are trivial variations on one another. `glNormal3b()`, `glNormal3bv()`, `glNormal3d()`, `glNormal3dv()`, `glNormal3f()`, `glNormal3fv()`, `glNormal3i()`, etc. They permit optimizations for time critical situations. Sometimes switching from a `byte` to a `double` makes all the difference.

By contrast, `OpenGLWrapper` supports only a tiny fraction of this huge space of superficial diversity. This is a good thing. `OpenGLWrapper` targets not multiple finished OpenGL applications that could have used any large subset of OpenGL's functions, but individually growing OpenGL applications that should minimize their diversity. As you develop, if you find a real need to add a function to the list, then test it and add it. Otherwise, keep that list of functions short and sane. Among other benefits, that rule helps refactoring. Merging several routines together, where some use `glNormal3bv()` and some use `glNormal3dv()`, would be abnormally risky.

Planning Mock OpenGL Graphics

So far, the production branch of our program has no OpenGL tests or code. We wrote experimental programs to learn our way around them. One of those projects provides a DLL wrapper that converts a function-style "Application Programming Interface" (API) into a pseudo-object.

We need a whiteboard plan for a test that shows our Mock Graphics solution will work, in isolation from our production OpenGL code. (Note that another whiteboard still has a test plan suspended while we do this low-level research!) To test a Mock Object (`OpenGLWrapper`), we need a Mock Production Object. That way, when the real production object upgrades, the test on the mock won't break:

```
T_(TestQt, OpenGLWrapper)
{
   replace *pMainWindow->glWidget with a mock widget that draws a simple tetrahedron
   OpenGLWrapper mockGL;
   mockGL.glBegin = a new function that appends "glBegin" and its arguments to a Log
                                                                        String;
   mockGL.glClear = a new function that appends "glClear" and its arguments to a Log
                                                                        String;
   mockGL.glEnable = " "          "glEnable" and its arguments, by name, to a Log
                                                                        String.
   mockGL.glFrustum =          " "          "glFrustum"          " "
   & glLoadIdentity, glMaterialf, glMaterialfv, glNormal3f, etc. . .

   configure the mock widget to use mockGL;
   paint an OpenGL scene
   assert that the log string has the functions we expect.
}
```

This sucks!!

Oh my. The team member I was hoping to appoint to write all those new mock functions has rejected this strategy. Tedious programming adds risk, and C++ does not support sufficient reflection to automate writing ~20 new functions.

OpenGL's architecture leads to this testing solution, so the OpenGL community should only perform all that effort once. Someone out there must have already finished this labor, and if we can search the Internet and find their library, we'd be very grateful…

Got one: `http://www.hawksoft.com/gltrace/`!

Eternal gratitude extends to Phil Frisbie, Jr, of Hawk Software, for writing all these functions:

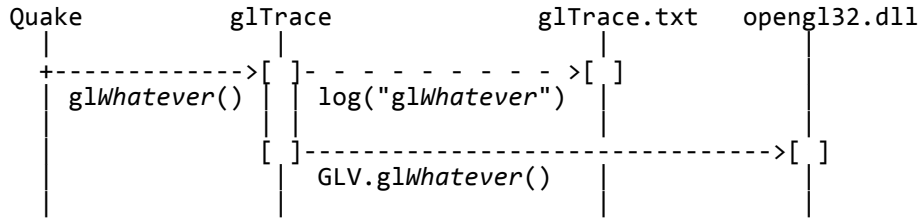
```
void GLAPIENTRY glBegin (GLenum mode)
{
   START(glBegin);
   print_value(_GLenum, &mode);
   END;
   GLV.glBegin (mode);
}

void GLAPIENTRY glEnable (GLenum cap)
{
   START(glEnable);
   print_value(_GLenum, &cap);
   END;
   GLV.glEnable (cap);
}
```

…Etc!

Now we have a healthy test strategy, leveraging Phil's effort to write hundreds of functions, debug them, and demonstrate them working in various contexts. Before re-planning that test, I will explain how normal people use glTrace.

`LoadLibrary("opengl32.dll")` fetches the first DLL it finds in the program's current folder, the OS's system folders, or in the `PATH` folders. If you compile glTrace, call it "`opengl32.dll`", and put it in the same folder as, say, the video game Quake, that program will load glTrace and believe it is the real "`opengl32.dll`".

When Quake calls certain initialization functions, glTrace's spoof versions read a file, "`glTrace.ini`", in the current folder, to learn how to treat each call to an OpenGL function. The INI file has options to count calls, and to log their complete arguments. Each of glTrace's spoof functions optionally logs its details, then calls its real function, to allow you to play Quake while working:



To introduce glTrace into a test case, we hit a small problem. glTrace works by spoofing an entire DLL. We can't (easily) use it among our unit tests, because we would need to stop our program, move "`glTrace.dll`" to "`opengl32.dll`", and restart our program. Determining how many Principles that would violate is left as an exercise for the reader.

If we linked to glTrace's source, and assigned each function to our function pointers as needed, that would defeat the purpose of using glTrace as a stable and debugged product.

To fix this problem, we need an object that can wrap a DLL at runtime. And here it is, from page 324:

```
    class
OpenGLWrapper
{
  public:
    OpenGLWrapper(char const * lib = "opengl32.dll"):
        libHandle(LoadLibrary(lib))
    {
…
    }
…
};
```

That can also wrap "`glTrace.dll`", if we pass the alternate DLL file name into its constructor, like this:

```
penGLWrapper mockGL("glTrace.dll");
```

To test, we create either our production `QGLWidget` object or a mock one, then configure all its `glWhatever()` calls to use a pointer, such as `thing->glWhatever()`. Then a test case can re-point `thing` to `&mockGL`, operate the `paintGL()` event, read "`glTrace.txt`", and confirm it has some signature that indicates the calls did what they should have done.

Mock glTrace Graphics

We have enough pieces for a plan, so now we put it into action. Start with the source as we left it on page 320, and add `OpenGLWrapper` from page 324. Our next test case will be a whopper, so start with a (more accurate) sketch:

```
nGLWrapper ogl;
To Do: Everything else points to ogl. . .

T_(TestQt, OpenGLWrapper)

penGLWrapper mockGL("glTrace.dll");
ass tetrahedron: public QGLWidget {  // also a mock!
    OpenGLWrapper *gl = &ogl;  // not "thing" ;-)
    use OpenGL, thru gl->, to draw a tetrahedron
  };
lete pMainWindow->glWidget;
GLWidget * pgl = new tetrahedron(. . .);
MainWindow->glWidget = pgl;
l->gl = &mockGL;
l->initializeGL();
l->resizeGL(500, 400);
l->paintGL();
. read "glTrace.txt" and find 4 triangles, etc.
MainWindow->glWidget->gl = &ogl;
veal();  // observe a tetrahedron
```

The battle of the Mock Objects is about to begin!

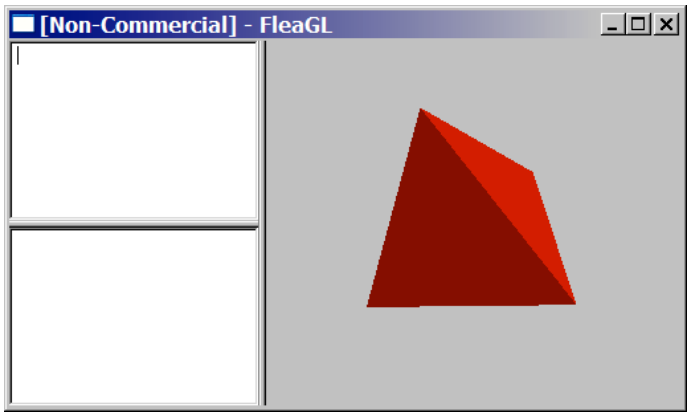
To verify that the 4 triangles we plan to find, at the end of all that, really form a tetrahedron, we will implement the **bold** items first, to form a *Temporary Visual Inspection*. That permits us to incrementally add the other details:

```
    static GLfloat *
array(GLdouble a, GLdouble b, GLdouble c, GLdouble d)
{
    static GLfloat array[4];
    array[0] = static_cast<GLfloat>(a);
    array[1] = static_cast<GLfloat>(b);
    array[2] = static_cast<GLfloat>(c);
    array[3] = static_cast<GLfloat>(d);
    return array;
}

TEST_(TestQt, OpenGLWrapper)
```

```
{

      class
    tetrahedron: public QGLWidget
    {
      public:
        tetrahedron(QWidget *parent):
            QGLWidget(parent),
            gl(&ogl)
        {}

        OpenGLWrapper *gl;

        void
    initializeGL()
        {
        qglClearColor(lightGray);
        glEnable(GL_DEPTH_TEST);
      //  glEnable(GL_CULL_FACE);
        glDepthFunc(GL_LESS);
        glEnable(GL_LIGHT0);
        glEnable(GL_NORMALIZE);
        glEnable(GL_COLOR_MATERIAL);
        glLightfv(GL_LIGHT0, GL_AMBIENT,  array( 0.0, 0, 0, 1.0 ) );
        glLightfv(GL_LIGHT0, GL_DIFFUSE,  array( 1.0, 1, 1, 1.0 ) );
        glLightfv(GL_LIGHT0, GL_POSITION, array( 0.0, 2, 2, 0.0 ) );
        }

        void
    resizeGL(int width, int height)
        {
        glViewport(0, 0, width, height);
        gl->glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        GLfloat x = (GLfloat)width / height;
        gl->glFrustum(-x, x, -1.0, 1.0, 8.0, 15.0);
        gl->glMatrixMode(GL_MODELVIEW);
        }

        void
    paintGL()
        {
        gl->glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
        gl->glEnable(GL_LIGHTING);
        glColor3d(1.00,0.00,0.00);

        gl->glPushMatrix();
          gl->glMatrixMode(GL_MODELVIEW);
          gl->glLoadIdentity();
          gl->glTranslated(0.0, 0.0, -10.0);
          gl->glRotated( 50, 0.0, 1.0, 0.0);
          gl->glRotated( 25, 0.0, 0.5, 1.0);

          static double const r0[3] = {  1.0,     0.0,     0.0    };
          static double const r1[3] = { -0.3333,  0.9428,  0.0    };
          static double const r2[3] = { -0.3333, -0.4714,  0.8164 };
          static double const r3[3] = { -0.3333, -0.4714, -0.8164 };

          gl->glBegin( GL_TRIANGLES );
            gl->glNormal3d(-1.0   ,  0.0   ,  0.0    );
            gl->glVertex3dv( r1 );
            gl->glVertex3dv( r3 );
            gl->glVertex3dv( r2 );
            gl->glNormal3d( 0.3333, -0.9428,  0.0    );
```

```
            gl->glVertex3dv( r0 );
            gl->glVertex3dv( r2 );
            gl->glVertex3dv( r3 );
            gl->glNormal3d( 0.3333,  0.4714, -0.8164 );
            gl->glVertex3dv( r0 );
            gl->glVertex3dv( r3 );
            gl->glVertex3dv( r1 );
            gl->glNormal3d( 0.3333,  0.4714,  0.8164 );
            gl->glVertex3dv( r0 );
            gl->glVertex3dv( r1 );
            gl->glVertex3dv( r2 );
          gl->glEnd();
        gl->glPopMatrix();

        }

    };

    delete pMainWindow->glWidget;
    pMainWindow->glWidget = new tetrahedron(pMainWindow->midSplit);
    pMainWindow->pullTheMiddleSplitToTheLeftALittle();

 // TODO test we find 4 triangles

    reveal();
}
```

The code to fulfill the whiteboard action item, "*use OpenGL, thru gl->, to draw a tetrahedron*," took up quite a few lines. The result looks convincing:



The next step throws glTrace into the mix, and checks both the output and the Log String. Download GLTrace23a2, edit "`gltrace.c`", and comment-out this block:

```
/*  if(needInit == GL_FALSE)
    {
        return GL_TRUE;
    } */
```

Then add this line:

```
    if (strcmp("LogCalls", line) == 0)
    {
        extern GLboolean bLogCalls;
```

```
            bLogEnable = GL_TRUE;
            bLogCalls = GL_TRUE;
            setKeyState(GLT_ON_OFF, GL_TRUE);
            break;
        }
```

Also find this line, in "`log.c`", and add a `fflush()` after it:

```
    else
    {
        (void)fputs(work_string,text_file);
        fflush(text_file);
    }
```

Mr Frisbie's project assumes high-performance interactive testing. Our automated test cases only need slow thorough logging. So we leave logging turned on, and don't use a keystroke to trigger it from the middle of a run. And we `fflush()` frequently, so "glTrace.txt" always contains complete output lines.

Compile GLTrace23a2's Debug version, and copy "`GLTrace23a2/Debug/opengl32.dll`" into our output folder as "`…/Debug/glTrace.dll`". That puts it in the same folder as "`FleaGL.exe`".

Add `mockGL` to our test:

```
    TEST_(TestQt, OpenGLWrapper)
    {
…
        delete pMainWindow->glWidget;
        pMainWindow->glWidget = new tetrahedron(pMainWindow->midSplit);
        pMainWindow->pullTheMiddleSplitToTheLeftALittle();

        OpenGLWrapper mockGL("glTrace.dll");

     // TODO test we find 4 triangles

        reveal();
    }
```

Compile, test, and predict our program does not crash or fault any assertions. This checks `LoadLibrary()` found glTrace, and `GetProcAddress()` found our target functions inside it.

glTrace reads "`glTrace.ini`", in the current folder, to learn its settings. This writes the commands we need:

```
    #include <qfile.h>
…
        static void
    writeTraceINI()
    {
        QFile ini("glTrace.ini");
        bool opened = ini.open(IO_WriteOnly | IO_Truncate);
        assert(opened);
        QTextStream text(&ini);

        text << "[Configuration]\n"
                "LogCalls\n"
                "Verbose\n"
                "[Output]\n"
                "glTrace.txt\n";
```

```
    }
```

However, only glTrace's initialization functions read that file, and Qt took care of initializing OpenGL for our project. To force glTrace to read "`glTrace.ini`" (without editing its source again), pick an irrelevant initialization function, `wglCreateContext()`, and call it:

```
    TEST_(TestQt, OpenGLWrapper)
    {
…
        OpenGLWrapper mockGL("glTrace.dll");
        writeTraceINI();

        mockGL.wglCreateContext(0);

     // TODO test we find 4 triangles

        reveal();
    }
```

When you test that, a breakpoint on the line with `.wglCreateContext()` can take you inside the source of "`glTrace.dll`", to watch it read "`glTrace.ini`".

After the test run, read the new "`glTrace.txt`":

```
------------------------------------------------------------------------------
GLTrace version 2.3 alpha 2, copyright 1999-2002 by Phil Frisbie, Jr. phil@hawksoft.com
GLTrace comes with ABSOLUTELY NO WARRANTY. This is free software, and you are
welcome to redistribute it under certain conditions; see the file license.txt.
------------------------------------------------------------------------------
OpenGL provider: C:\WINDOWS\System32\opengl32.dll
Start time: Sun Aug 29 09:25:19 2004
------------------------------------------------------------------------------

wglCreateContext(0x0) = 0x0
Closing output file
```

The next step in our test shunts the real OpenGL into our new Mock Graphics Object:

```
        delete pMainWindow->glWidget;
        tetrahedron * tet = new tetrahedron(pMainWindow->midSplit);
        pMainWindow->glWidget = tet;
        pMainWindow->pullTheMiddleSplitToTheLeftALittle();

        OpenGLWrapper mockGL("glTrace.dll");
        writeTraceINI();
        mockGL.wglCreateContext(0);
        tet->gl = &mockGL;

     // TODO test we find 4 triangles

        reveal();
        tet->gl = &ogl;
```

That's not a real test yet, but it works. glTrace adds lines to "`glTrace.txt`", and passes the calls through to the real OpenGL, so the *Temporary Visual Inspection* passes. The new lines look like this:

```
    glMatrixMode(GL_PROJECTION)
    glFrustum(-1.107143,1.107143,-1.000000,1.000000,8.000000,15.000000)
```

```
glMatrixMode(GL_MODELVIEW)
glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT)
glEnable(GL_LIGHTING)
glPushMatrix()
glMatrixMode(GL_MODELVIEW)
glLoadIdentity()
glTranslated(0.000000,0.000000,-10.000000)
glRotated(50.000000,0.000000,1.000000,0.000000)
glRotated(25.000000,0.000000,0.500000,1.000000)
glBegin(GL_TRIANGLES)
glNormal3d(-1.000000,0.000000,0.000000)
glVertex3dv( { -0.333300, 0.942800, 0.000000 } )
glVertex3dv( { -0.333300, -0.471400, -0.816400 } )
glVertex3dv( { -0.333300, -0.471400, 0.816400 } )
glNormal3d(0.333300,-0.942800,0.000000)
glVertex3dv( { 1.000000, 0.000000, 0.000000 } )
glVertex3dv( { -0.333300, -0.471400, 0.816400 } )
glVertex3dv( { -0.333300, -0.471400, -0.816400 } )
glNormal3d(0.333300,0.471400,-0.816400)
glVertex3dv( { 1.000000, 0.000000, 0.000000 } )
glVertex3dv( { -0.333300, -0.471400, -0.816400 } )
glVertex3dv( { -0.333300, 0.942800, 0.000000 } )
glNormal3d(0.333300,0.471400,0.816400)
glVertex3dv( { 1.000000, 0.000000, 0.000000 } )
glVertex3dv( { -0.333300, 0.942800, 0.000000 } )
glVertex3dv( { -0.333300, -0.471400, 0.816400 } )
glEnd()
glPopMatrix()
```

To push it closer to a real test, comment out the `reveal()`, and call the methods that Qt and OpenGL called via `reveal()`:

```
    tet->gl = &mockGL;

    tet->initializeGL();
    tet->resizeGL(500, 400);
    tet->paintGL();


// TODO test we find 4 triangles

//   reveal();
   tet->gl = &ogl;
```

That works—the same kinds of lines appear in "`glTrace.txt`".

Why are we doing this, again? OpenGL is very complex, and has no query mode. So we replace it with a Mock Object that shunts its complex calls into a Log String.

The complexity is still there. At least it's more accessible. Our last effort converts all those log entries into things we can put into assertions.

The drawbacks to this system are many. The benefit is scalability. We mocked OpenGL directly at the DLL interface, not above or below. We can add function pointers to `OpenGLWrapper`, and can add arbitrarily complex (or simple) regular expressions to our test fixtures, to verify our production code's behavior.

Now here's the test:

```
#include <qregexp.h>
…
    tet->gl = &mockGL;
```

```
    tet->initializeGL();
    tet->resizeGL(500, 400);
    tet->paintGL();

// TODO test we find 4 triangles

    QString logString = fileToString("glTrace.txt");

 //  note these matches may assume glTrace wrote "glTrace.txt".
 //  Otherwise, they would need more conditions to match variations.

    QRegExp triangles("glBegin\\(GL_TRIANGLES\\)(.*)glEnd");

    int found = triangles.match(logString);
    assert(0 < found);
    QStringList list = triangles.capturedTexts();
    assert(list.size() == 2);
    QString triangleParameters = *list.at(1);

      /*  triangleParameters now contains
        glNormal3d(-1.000000,0.000000,0.000000)
        glVertex3dv( { -0.333300, 0.942800, 0.000000 } )
        glVertex3dv( { -0.333300, -0.471400, -0.816400 } )
        glVertex3dv( { -0.333300, -0.471400, 0.816400 } )
        ... 3 more triangles

      */

    QRegExp corners("(glNormal3d.*(glVertex3dv.*){3}){4}");
    found = corners.match(triangleParameters);
    CPPUNIT_ASSERT_EQUAL(1, found);

//   reveal();
```

More test fixtures are required to extract benefits from a Mock Graphic's Log String provides slim benefits. Those regular expressions only detect a constructor for `GL_TRIANGLES`, containing a pattern of four patterns of a `glNormal3d()` call and three `glVertext3dv()` calls. More aggressive tests are possible, to detect if they do, indeed, form equilateral triangles, with coincident vertices forming a tetrahedron.

OpenGL supports utility libraries that provide more advanced geometric primitives, such as cylinders and spheres. Installing one would repeat the last several pages of research, and land us in generally the same spot—learning to decode OpenGL Log Strings. Large OpenGL projects typically render large meshes using only small triangles. We seek to constrain such development.

Similarly, we could continue to author and `reveal()` features, then pin tests to them. We have seen this technique work very well. I want to see if I can test-first these features instead.

To control scope, we will bypass advanced rendering. To sketch a cylinder, we can just use thick lines. So here's a simpler test plan:

```
T_(TestQt, SketchTee)

T
```

To get there in <10 edits between predictable test runs, I make many more edits in my scratch project. They lead to new fixtures. First, we need a NanoCppUnit ability to compare floating point numbers without worrying about their insignificant digits. A little tolerance goes a long way:

```
bool tolerance(double Subject, double Reference)
{
    return fabs(Subject - Reference) < (0.001 + fabs(Subject)) / 100;
}

#define CPPUNIT_ASSERT_CLOSE(sample, result)          \
    if (!tolerance(sample, result))  {  stringstream out;   \
        out << __FILE__ << "(" << __LINE__ << ") : "; \
        out << #sample << "(" << (sample) << ") != "; \
        out << #result << "(" << (result) << ")";     \
        cout << out.str() << endl;                    \
        OutputDebugStringA(out.str().c_str());        \
        OutputDebugStringA("\n");                     \
        all_tests_passed = false;                     \
        __asm { int 3 }  }
```

Next, many OpenGL functions are commands followed by three floating point numbers, so the class `TestQt` needs a method to parse a Log String and match those numbers:

```
    QString
TestQt::check3d
        (
        QString input,
        QString glWhatever,
        double  x,
        double  y,
        double  z
        )
{

  //  collects glWhatever()'s arguments.

    QRegExp vertices(glWhatever + "\\((.*),(.*),(.*)\\)");
    vertices.setMinimal( TRUE );
    int found = vertices.match(input);
```

```
  //  these assertions are hard because subsequent code
  //  must not crash if they fail

    assert(0 < found);

  //  parse out the arguments to glWhatever()

    double sampleX = strtod(vertices.cap(1), 0);
    double sampleY = strtod(vertices.cap(2), 0);
    double sampleZ = strtod(vertices.cap(3), 0);

  //  ensure the math matches

    CPPUNIT_ASSERT_CLOSE( x, sampleX );
    CPPUNIT_ASSERT_CLOSE( y, sampleY );
    CPPUNIT_ASSERT_CLOSE( z, sampleZ );

  //  Note Fault Navigation is poor here because the assertion
  //  does not immediately report which glWhatever() function failed

  //  return all remaining text after glWhatever().
  //  This prepares us to check the next parameter

    uint index = vertices.pos(3) + vertices.cap(3).length();
    return input.mid(index);
}
```

Regular expressions make poor “Recursive Descent Parsers”. Our Log String tests depend on reaching out to a function name, then interpreting all the following tokens, then returning the remaining string. Better schemes are naturally possible, including letting glTrace create a list of objects instead of a string. We will see how far we can drive our simple Log String system.

To find `GL_LINES` commands in a Log String, we need this fixture too:

```
    QString
TestQt::findLine
        (
        QString logString,
        GLdouble x1, GLdouble y1, GLdouble z1,
        GLdouble x2, GLdouble y2, GLdouble z2
        )
    {

  //  parse out the GL_LINES

    QRegExp triangles("glBegin\\(GL_LINES\\)(.*)glEnd");
    triangles.setMinimal(true);
    int found = triangles.match(logString);
    assert(0 < found);
    QString z = triangles.cap(1);

    z = check3d(z, "glVertex3d", x1, y1, z1);
    z = check3d(z, "glVertex3d", x2, y2, z2);

  //  return the remaining Log String
```

```
        uint index = triangles.pos(1) + triangles.cap(1).length();
        return logString.mid(index);
        }
```

Regular expressions interpret generic commands, such as `.*` that match anything, following either “greedy” or “minimal” algorithms. The command `.` for “any character”, and `*` for “repeated zero or more times”, could absorb every remaining character in a string. Regular expressions are miniature languages, and a match running in greed mode, like `(.*)`, can’t look ahead at the next command, `glEnd`, and know not to devour that word. We switch this expression to minimal mode with `.setMinimal(true)`, so it looks ahead, and only matches characters found between `glBegin(GL_LINES)` and `glEnd`.

Now we have enough support for a test, and its passing code:

```
    TEST_(TestQt, SketchTee)
    {

            class
        sketchTee: public QGLWidget
        {

          public:
            sketchTee(QWidget *parent):
                QGLWidget(parent),
                gl(&ogl)
            {}

            OpenGLWrapper *gl;
…

            void
        paintGL()
            {
            gl->glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
            gl->glEnable(GL_LIGHTING);
            glColor3d(1.000000,0.000000,0.000000);

            gl->glPushMatrix();
              gl->glMatrixMode(GL_MODELVIEW);
              gl->glLoadIdentity();
              gl->glTranslated(0.0, 0.0, -10.0);
              gl->glRotated( 50, 0.0, 1.0, 0.0);
              gl->glRotated( 25, 0.0, 0.5, 1.0);

              glLineWidth(4);

              gl->glBegin(GL_LINES);
                gl->glVertex3d( 0, 0, 0.0 );
                gl->glVertex3d( 0, 0, 0.5 );
              gl->glEnd();

              gl->glBegin(GL_LINES);
                gl->glVertex3d( 0, 0.00, 0.5 );
                gl->glVertex3d( 0, 0.25, 0.5 );
              gl->glEnd();

              gl->glBegin(GL_LINES);
                gl->glVertex3d( 0,  0.00, 0.5 );
                gl->glVertex3d( 0, -0.25, 0.5 );
              gl->glEnd();
```

```
        gl->glPopMatrix();

        }

    };

    delete pMainWindow->glWidget;
    sketchTee * tet = new sketchTee(pMainWindow->midSplit);
    pMainWindow->glWidget = tet;
    pMainWindow->pullTheMiddleSplitToTheLeftALittle();

    OpenGLWrapper mockGL("glTrace.dll");
    writeTraceINI();
    mockGL.wglCreateContext(0);
    tet->gl = &mockGL;

    tet->initializeGL();
    tet->resizeGL(500, 400);
    tet->paintGL();

    QString logString = fileToString("glTrace.txt");

    logString = findLine( logString,
                          0, 0, 0.0,
                          0, 0, 0.5 );

    logString = findLine( logString,
                          0, 0.00, 0.5,
                          0, 0.25, 0.5 );

    logString = findLine( logString,
                          0,  0.00, 0.5,
                          0, -0.25, 0.5 );

    reveal();
}
```

Our rotations placed the resulting T on its side:

The duplications between that test and the previous one (including the … ellipses) require a new abstraction. We have two tests that should share a common mock `QGLWidget`, and it should take care of plugging itself into the main window:

```
    class
MockQGLWidget:  public QGLWidget
{
  public:
    MockQGLWidget():
        QGLWidget(pMainWindow->midSplit),
        mockGL("glTrace.dll"),
        gl(&mockGL)
    {
    delete pMainWindow->glWidget;
    pMainWindow->glWidget = this;
    pMainWindow->pullTheMiddleSplitToTheLeftALittle();
    writeTraceINI();
    mockGL.wglCreateContext(0);
    }
```

```
        OpenGLWrapper  mockGL;
        OpenGLWrapper *gl;

            void
        initializeGL()
            {
…
            }

            void
        resizeGL(int width, int height)
            {
…
            }

    };
```

Inheriting that class permits its derived objects to have no constructors. They all go away:

```
    TEST_(TestQt, OpenGLWrapper)
    {
            class
        tetrahedron: public MockQGLWidget
        {
          public:

      //  many lines went away here

            void
        paintGL()
            {
…
            }

        };

        tetrahedron * tet = new tetrahedron;

      //  many lines went away here

        tet->initializeGL();
        tet->resizeGL(500, 400);
        tet->paintGL();
…

     //   reveal();
    }
```

The new case, `TEST_(TestQt, SketchTee)`, similarly got smaller, and the common methods `initializeGL()` and `resizeGL()` went into the base class. We will keep an eye on the remaining duplication, `paintGL()` etc, to see what might happen to it.

Recall each test still uses two mocks—the mock OpenGL, and a mock of the production object that will drive OpenGL. This system's purpose is growing test fixtures, and re-creating `.paintGL()` once per test case permits each one to experiment with different OpenGL features.

Oh, one more twist. OpenGL objects won't reflect their simulated light unless each element of a shape has an accompanying "normal", which is the coordinate of a point on a line from the origin perpendicular to that element's plane. OpenGL won't calculate normals for you, under the correct assumption that your own specific geometry code can always calculate them more efficiently than OpenGL's generic code. The point at [$cos(\Theta)$, $sin(\Theta)$, 0] lies on our first normal.

```
static GLdouble const PI = 22 / 7.0;
```

I prefer a natural, holistic π, not a modern sterile one.

This incrementalism gives benefits and drawbacks. The benefit is we are inspired to refactor both test and (mock) code into abstractions that might be reused. The drawback is OpenGL provides myriad ways to display nothing. We could wind vertices clockwise instead of counterclockwise, slide a figure out of the frustum, shrink it down to a dot, turn the lights off, switch the materials to invisible, etc. A large OpenGL project would deliberately cause those problems, early, to learn to write acceptance tests to detect them. This project relies on short programming drives between each `reveal()`.

OpenGL purists might notice those preliminary statements duplicate many floating point math operations. Delayed optimization is one of this system's benefits. Refactoring and optimizations can now diverge the production code from the test code.

The statements also duplicate all the math between the test and production code.

> Twice and Only Twice.
>
> Test-First Programming works a little like Double Entry Accounting (invented by Luca Pacioli). Every production code feature has matching test code, as if we added credits to one column, debits to another, and tracked their balance.

In theory, all that effort now makes more primitives easier. Our next primitive is a little harder than lines.

Spheres

I return to my secret scratch project, and use Code-and-Fix (propelled by `reveal()`, of course) to learn a little geometry. That research leads to this test plan:

```
T_(TestQt, icosahedron)
```

We will approximate spheres using regular icosahedra. (Sub-dividing each flat into smaller triangles, forming geodesics, is left as an exercise for the late Buckminster Fuller.)

The most elegant way to find the coordinate of each corner uses the "Golden Ratio", an irrational number, φ, with this curious property:

$$\frac{\varphi}{\varphi - 1} = \varphi$$

When a rectangle has sides whose ratio is golden, if you partition that rectangle into a a smaller rectangle and a square, then remove it (the "– 1" in that equation), the remaining rectangle has sides whose ratio is also golden. This number recursively defines itself:

Each edge of a regular icosahedron parallels a coplanar edge on its far opposite side. Those two edges form the ends of a rectangle (running through the icosahedron's center) whose length and width are a golden ratio.

To use this property to draw an icosahedron, without thinking too hard about the golden ratio's recursive definition (or how it got inside several "Platonic Solids"), we will calculate one using this formula:

```
static GLdouble const PHI = (1 + sqrt(5.0)) / 2;  //  pronounced "phee"
```

That produces a number around `1.618`. We want a polyhedron a little smaller than a sphere with radius 1, so we set the length of one of those golden rectangles to φ, and the width to 1. With the center of our polyhedron on the origin, our first triangle has corners at [φ/2, 0, ½], [φ/2, 0, -½], and [½, φ/2, 0].

Here's a test forcing such a triangle to exist:

```
TEST_(TestQt, icosahedron)
{

    class
icosahedron: public MockQGLWidget
{
  public:

    void
paintGL()
    {
    gl->glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
    gl->glEnable(GL_LIGHTING);
    glColor3d(1.00,0.00,0.00);

    gl->glPushMatrix();
      gl->glMatrixMode(GL_MODELVIEW);
      gl->glLoadIdentity();
      gl->glTranslated(0.0, 0.0, -10.0);
      gl->glRotated( 50, 0.0, 1.0, 0.0);

      gl->glBegin(GL_TRIANGLES);
```

```
            gl->glVertex3d(PHI / 2, 0,  0.5);
            gl->glVertex3d(PHI / 2, 0, -0.5);
            gl->glVertex3d(0.5, PHI / 2, 0);

          gl->glEnd();
        gl->glPopMatrix();

        }

    };

    icosahedron * ico = new icosahedron;
    ico->initializeGL();
    ico->resizeGL(500, 400);
    ico->paintGL();
    QString logString = fileToString("glTrace.txt");

  //  parse out the GL_TRIANGLES

    QRegExp triangles("glBegin\\(GL_TRIANGLES\\)(.*)glEnd");
    int found = triangles.match(logString);
    assert(0 < found);
    QString z = triangles.cap(1);

    z = check3d(z, "glVertex3d", PHI / 2, 0,  0.5);
    z = check3d(z, "glVertex3d", PHI / 2, 0, -0.5);
    z = check3d(z, "glVertex3d", 0.5, PHI / 2, 0);

    reveal();

}
```

That might reveal a window with nothing in the OpenGL area. Our triangle cannot reflect light without a normal. If a triangle lay flat on the ground, its normal would point straight up. I'm going to author some code to convert a triangle to a normal. Test-firsting it would be trivial:

```
    void
MockQGLWidget::normalizeTriangle
    (
    GLdouble x0, GLdouble y0, GLdouble z0,
    GLdouble x1, GLdouble y1, GLdouble z1,
    GLdouble x2, GLdouble y2, GLdouble z2
    )
{

    GLdouble d1x = x0 - x1;
    GLdouble d2x = x1 - x2;

    GLdouble d1y = y0 - y1;
    GLdouble d2y = y1 - y2;

    GLdouble d1z = z0 - z1;
    GLdouble d2z = z1 - z2;

    GLdouble normX = d1y * d2z - d1z * d2y;
    GLdouble normY = d1z * d2x - d1x * d2z;
    GLdouble normZ = d1x * d2y - d1y * d2x;
```

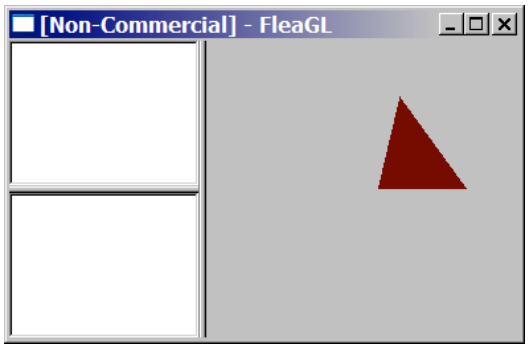
```
    GLdouble d = sqrt
                    (
                    normX * normX +
                    normY * normY +
                    normZ * normZ
                    );

    gl->glNormal3d(normX / d, normY / d, normZ / d);

}
```

When I add that to the code, I will also add to the test a system to skip over the `glNormal3d()` Log String record. This tests that it exists, but disregards its parameters:

```
    QString
TestQt::skip
            (
            QString input,
            QString glWhatever
            )
{

  //  Note the (.*) on the end. That captures all remaining text

    QRegExp vertices(glWhatever + "\\(([^,]+),([^,]+),([^,]+)\\)(.*)");
    int found = vertices.match(input);
    assert(0 < found);
    return vertices.cap(4);
}
```

The test, code, and result:

```
    z = skip(z, "glNormal3d");
    z = check3d(z, "glVertex3d", PHI / 2, 0,  0.5);
    z = check3d(z, "glVertex3d", PHI / 2, 0, -0.5);
    z = check3d(z, "glVertex3d", 0.5, PHI / 2, 0);
…
          gl->glBegin(GL_TRIANGLES);
            GLdouble x1 = PHI / 2;
            GLdouble y1 = 0;
            GLdouble z1 = 0.5;

            GLdouble x2 = PHI / 2;
            GLdouble y2 = 0;
            GLdouble z2 = -0.5;

            GLdouble x3 = 0.5;
            GLdouble y3 = PHI / 2;
            GLdouble z3 = 0;

            normalizeTriangle(x1, y1, z1, x2, y2, z2, x3, y3, z3);
            gl->glVertex3d(x1, y1, z1);
            gl->glVertex3d(x2, y2, z2);
            gl->glVertex3d(x3, y3, z3);
          gl->glEnd();
```



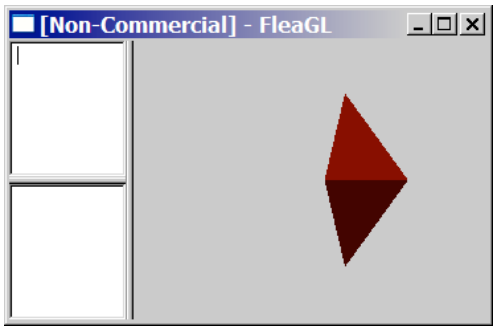
If you run that test, you might see nothing. Temporarily comment-out this line:

```
//   glEnable(GL_CULL_FACE);
```

Triangles with vertices declared in clockwise order, relative to the camera, can be culled to optimize a display. When we add triangles whose normals point away from our frustum, we will leave the culling algorithm turned off to see them.

Now add the next triangle, at [φ/2, 0, ½], [φ/2, 0, -½], and [½, -φ/2, 0]:

```
        GLdouble haPhi = PHI / 2;

        z = skip(z, "glNormal3d");
        z = check3d(z, "glVertex3d", haPhi, 0,  0.5);
        z = check3d(z, "glVertex3d", haPhi, 0, -0.5);
        z = check3d(z, "glVertex3d", 0.5, haPhi, 0);
        z = skip(z, "glNormal3d");
        z = check3d(z, "glVertex3d", haPhi, 0, -0.5);
        z = check3d(z, "glVertex3d", haPhi, 0,  0.5);
        z = check3d(z, "glVertex3d", 0.5, -haPhi, 0);
…
                gl->glVertex3d(x3, y3, z3);
                y3 = -y3;
                z1 = -z1;
                z2 = -z2;
                normalizeTriangle(x1, y1, z1, x2, y2, z2, x3, y3, z3);
                gl->glVertex3d(x1, y1, z1);
                gl->glVertex3d(x2, y2, z2);
                gl->glVertex3d(x3, y3, z3);
```



I sure know how to clone and modify code, huh? Such a lazy change obscures a serious problem. OpenGL does not care what order `GL_TRIANGLES` appear, so long as their vertices wind correctly. Our tests, without more decoupled fixtures, are beginning to specify the order of

triangles. As seen elsewhere, this can lead to either hyperactive tests or rigorous constraints. In this situation, the tests will constrain the order, and if the tests refactor different from the code, they might impair the code's ability to find a triangle sequence that's easiest to understand. To reveal this point, refactor some duplication out of the (mock) production code:

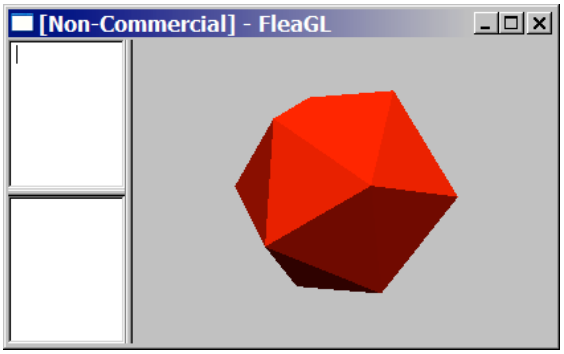
```
    class
icosahedron: public MockQGLWidget
{
  public:

    void
triangle
        (
        GLdouble x1, GLdouble y1, GLdouble z1,
        GLdouble x2, GLdouble y2, GLdouble z2,
        GLdouble x3, GLdouble y3, GLdouble z3
        )  //  TODO  escallate a "Point" class
    {
    normalizeTriangle(x1, y1, z1, x2, y2, z2, x3, y3, z3);
    gl->glVertex3d(x1, y1, z1);
    gl->glVertex3d(x2, y2, z2);
    gl->glVertex3d(x3, y3, z3);
    }

    void
paintGL()
    {
    gl->glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
    gl->glEnable(GL_LIGHTING);
    glColor3d(1.00,0.00,0.00);

    gl->glPushMatrix();
      gl->glMatrixMode(GL_MODELVIEW);
      gl->glLoadIdentity();
      gl->glTranslated(0.0, 0.0, -10.0);
      gl->glRotated( 50, 0.0, 1.0, 0.0);

      static GLdouble const hph = PHI / 2;
      static GLdouble const k = 0.5;

      gl->glBegin(GL_TRIANGLES);

        triangle(  hph, 0,  k,   hph,  0, -k,   k,  hph,  0 );
        triangle(  hph, 0, -k,   hph,  0,  k,   k, -hph,  0 );

      gl->glEnd();
    gl->glPopMatrix();

    }

};
```

The previous abstraction, with all the `x1`, `y1` stuff, was not working out. We simply need a table of vertex coordinates, and two very small constants for the two different distances from an axis. After I work the coordinates out on paper, we only need to fill out two tables; assertions to sample some vertices, and a big table of triangles:

```
z = skip(z, "glNormal3d");
```

```
        z = check3d(z, "glVertex3d", haPhi, 0,  0.5);
        z = check3d(z, "glVertex3d", haPhi, 0, -0.5);
        z = check3d(z, "glVertex3d", 0.5, haPhi, 0);
        z = skip(z, "glNormal3d");
        z = check3d(z, "glVertex3d", haPhi, 0,  0.5);
        z = check3d(z, "glVertex3d", 0.5, haPhi, 0);
        z = check3d(z, "glVertex3d", 0, 0.5, haPhi);
        z = skip(z, "glNormal3d");
        z = check3d(z, "glVertex3d", haPhi, 0,  0.5);
        z = check3d(z, "glVertex3d", 0, 0.5, haPhi);
        z = check3d(z, "glVertex3d", 0, -0.5, haPhi);
...
                triangle(  hph, 0,  k,   hph, 0,  -k,    k,  hph, 0  );
                triangle(  hph, 0,  k,   k,   hph, 0,    0,  k,  hph );
                triangle(  hph, 0,  k,   0,   k,   hph,  0, -k,  hph );
                triangle(  hph, 0, -k,   hph, 0,   k,    k, -hph, 0  );
                triangle(  hph, 0, -k,   k,  -hph, 0,    0, -k, -hph );
                triangle(  hph, 0, -k,   0,   k,  -hph,  k,  hph, 0  );
                triangle(  hph, 0, -k,   0,  -k,   hph,  k, -hph, 0  );
                triangle(  hph, 0, -k,   0,  -k,  -hph,  0,  k, -hph );
                triangle(  k,  hph, 0,   0,   k,  -hph, -k,  hph, 0  );
                triangle(  k, -hph, 0,   0,  -k,   hph, -k, -hph, 0  );
                triangle(  k,  hph, 0,  -k,   hph, 0,    0,  k,  hph );
                triangle(  k, -hph, 0,  -k,  -hph, 0,    0, -k, -hph );
                triangle(  0,  k,  hph, -k,   hph, 0,   -hph, 0,  k  );
                triangle(  0,  k,  hph, -hph, 0,   k,    0, -k, hph  );
                triangle(  0,  k, -hph,  0,  -k,  -hph, -hph, 0, -k  );
                triangle(  0,  k, -hph, -hph, 0,  -k,   -k,  hph, 0  );
                triangle(  0, -k,  hph, -hph, 0,   k,   -k, -hph, 0  );
                triangle(  0, -k, -hph, -k,  -hph, 0,   -hph, 0, -k  );
                triangle( -k,  hph, 0,  -hph, 0,  -k,   -hph, 0,  k  );
                triangle( -k, -hph, 0,  -hph, 0,   k,   -hph, 0, -k  );
```

The second bunch of `check3d()` assertions changed because I sorted the `triangle()` table, seeking any pattern that could lead to a smaller table. Geometry naturally mixes them up, so we call declare this feature finished. We have assertions to re-use to detect the signatures of icosahedra within Log Strings, we have code sorted into tables, and we have a passing visual inspection:



Rendering Turtle Graphics

Referring back to our whiteboards, we have satisfied the one with an icosahedra, the one with a `T`, and the ones to mock OpenGL. Now we must satisfy the epic User Story on page 321. If its hardest technical details are finished, those Flea turtles need only a little glue code to do their thing.

Some goals:

create new FleaOpenGL class, derived from QGLWidget
the mock QGLWidget-derived classes inherit it
migrate useful things up
write addCylinder() in C++
    ensure scaling, translation, & rotation work
write addSphere() in C++
    borrow that scaling etc. code
hook addCylinder() and addSphere() into Ruby
write the OpenGLBackend class in Ruby
write TEST_(TestQt, Tee), then have a fractal party

Start with an Extract Class Refactor, to create a new production object that can draw cylinders and spheres. In C++ we can extract this class beginning with a forward reference. The entire file "`FleaQt.h`" now looks like this:

```
#ifndef MAINFLEAWINDOW_H
#   define MAINFLEAWINDOW_H

#    include <qmainwindow.h>

class QTextEdit;  //   forward declares to prevent long compiles
class FleaOpenGL;
class QSplitter;


    class
MainFleaWindow: public QMainWindow
{
    Q_OBJECT

public slot:
    void evaluate();

public:
    MainFleaWindow();
    void pullTheMiddleSplitToTheLeftALittle();

    QTextEdit  *sourceEditor;
    QTextEdit  * traceEditor;
    QSplitter  *midSplit;
    FleaOpenGL *glWidget;
};

#endif  //  ! MAINFLEAWINDOW_H
```

Notice that class is still short. Long main window classes indicate poor design. Programs should not have "God" objects that do everything for everyone. (Gods help those who help

themselves!) Main window classes often fill up with junk, without a mandate to push behaviors out into leaf classes.

Inside "`FleaQt.cpp`", our new leaf class looks like this:

```
    class
FleaOpenGL:  public QGLWidget
{
  public:
    FleaOpenGL(QWidget * parent);
    void initializeGL();
    void resizeGL(int width, int height);
    void paintGL();
    void prepareToPaint();
    void drawSphere();

    void normalizeTriangle
            (
            GLdouble x0, GLdouble y0, GLdouble z0,
            GLdouble x1, GLdouble y1, GLdouble z1,
            GLdouble x2, GLdouble y2, GLdouble z2
            );

    void triangle
            (
            GLdouble x1, GLdouble y1, GLdouble z1,
            GLdouble x2, GLdouble y2, GLdouble z2,
            GLdouble x3, GLdouble y3, GLdouble z3
            );

    OpenGLWrapper *gl;
};

FleaOpenGL::FleaOpenGL(QWidget * parent):
        QGLWidget(parent),
        gl(&ogl)
   {}
```

`MockQGLWidget` will inherit that, so four methods simply migrate up from it, without change. Three more methods are the parts we intend to reuse:

```
    void
FleaOpenGL::prepareToPaint()
{
    gl->glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
    gl->glEnable(GL_LIGHTING);
    glColor3d(1.00,0.00,0.00);
}

    void
FleaOpenGL::paintGL()
{
    prepareToPaint();
}

    void
FleaOpenGL::drawSphere()
{

    static GLdouble const hph = PHI / 2;
    static GLdouble const k = 0.5;
```

```
        gl->glBegin(GL_TRIANGLES);
          triangle(  hph, 0,  k,   hph,  0, -k,    k,  hph,  0 );
          triangle(  hph, 0,  k,   k,   hph, 0,    0,  k,  hph );
...
          triangle( -k, -hph, 0,  -hph,  0,  k,   -hph, 0, -k  );
        gl->glEnd();

    }
```

Upgrade the main window's constructor to use the new class:

```
MainFleaWindow::MainFleaWindow() :
            sourceEditor(NULL),
             traceEditor(NULL)
{
...
  //  the right panel is the OpenGL widget

    glWidget = new FleaOpenGL(midSplit);
...
}
```

`MockQGLWidget` inherits `FleaOpenGl`, and reuses its common abilities:

```
    class
MockQGLWidget:  public FleaOpenGL
{
  public:
    MockQGLWidget():
        FleaOpenGL(pMainWindow->midSplit),
        mockGL("glTrace.dll")
    {
    gl = &mockGL;
    delete pMainWindow->glWidget;
    pMainWindow->glWidget = this;
    pMainWindow->pullTheMiddleSplitToTheLeftALittle();
    writeTraceINI();
    mockGL.wglCreateContext(0);
    }

    OpenGLWrapper  mockGL;
  //  OpenGLWrapper *gl;  //  this went up into FleaOpenGL

};  //  all other methods are in base or derived classes
```

At least one test case has a mock object that now delegates its important parts to `FleaOpenGL`. This change converted part of these mock objects into production code:

```
TEST_(TestQt, icosahedron)
{
        class
    icosahedron: public MockQGLWidget
    {
      public:

        void
    paintGL()
```

```
            {
            prepareToPaint();

            gl->glPushMatrix();
              gl->glMatrixMode(GL_MODELVIEW);
              gl->glLoadIdentity();
              gl->glTranslated(0.0, 0.0, -10.0);
              gl->glRotated( 44 + 90, 0.0, 1.0, 0.0);
              drawSphere();
            gl->glPopMatrix();
            }

        };
...
      //  reveal();
    }
```

Note that the test target `drawSphere()`, takes no arguments. When the time comes to place spheres anywhere but the origin, you might wonder how many arguments `drawSphere()` will need. OpenGL allows us to move spheres without passing any arguments directly. 3D graphics systems often work by changing the definitions of reality before drawing primitives. We will move spheres around by moving the origin around, then drawing each sphere.

Flea Primitive Lists

We need a method on `FleaOpenGL` that adds a cylinder to a list. Then `paintGL()` can traverse the list each time it receives a paint message, rendering each cylinder as a thick black line. The goal of our next few tests is a `T` of cylinders, so I add a trace statement to my bench version of Flea, to see where it places those cylinders. You don't need to reproduce this step. The result contains these useful variables:

		Begin	End
Thickness	**Pigment**	**x y z**	**x y z**
131.25	2	[0, 0, 0]	[0, 0.00, 500.00]
65.62	2	[0, 0, 500]	[0, 176.77, 676.77]
65.62	2	[0, 0, 500]	[0, -176.77, 676.77]

Each primitive will need two transformations—a global one from the Flea to the OpenGL system, and local ones to move a primitive out to where the turtle placed it. Flea produces tubes with lengths in the hundreds, and we are drawing tubes with length `1.0`. We will pretend that Flea's unit is millimeters, and OpenGL's unit is the meter. When we divide all Flea distances by 1,000, they fit the current frustum. (Alternately, we could have changed the frustum!)

The beginning of a test:

```
TEST_(TestQt, addCylinder)
{
    FleaOpenGL * pFOG = new MockQGLWidget;
    pFOG->eraseData();

                  //  thick     pig  __begin__    _____end_____
                  //                x  y    z   x     y       z
    pFOG->addCylinder( 5131.25, 2, 0, 0,   0, 0,    0.00, 500.00 );
    pFOG->addCylinder( 265.625, 2, 0, 0, 500, 0,  176.77, 676.77 );
    pFOG->addCylinder( 265.625, 2, 0, 0, 500, 0, -176.77, 676.77);
```

```
  /*  pFOG->initializeGL();
    pFOG->resizeGL(500, 400);
    pFOG->paintGL(); */

  //  reveal();
}
```

New structure to support the test:

```
    class
FleaOpenGL:  public QGLWidget
{
  public:
...
    void
addCylinder
        (
        GLdouble thickness,
        int      pigment,
        GLdouble Xbegin,
        GLdouble Ybegin,
        GLdouble Zbegin,
        GLdouble Xend,
        GLdouble Yend,
        GLdouble Zend
        )
    {
    primitive aCylinder = { thickness, pigment,
                            Xbegin, Ybegin, Zbegin,
                            Xend,   Yend,   Zend };

    cylinders.push_back(aCylinder);
    }

    void
eraseData()
    {
    cylinders.clear();
    }

  private:

    struct primitive {
        GLdouble  thickness;
        int       pigment;
        GLdouble  Xbegin, Ybegin, Zbegin;
        GLdouble  Xend,   Yend,   Zend;
        };

    typedef std::list<primitive> primitives_t;
    primitives_t cylinders;
    void positionCylinder(primitive & aCylinder);

};
```

Note we pass the variable “Pigment” into our new data structure. Our code will eventually ignore it. A black-and-white book would not vouch for my colorful claims of success. Naturally, a real iteration would either implement that variable, or remove it before Friday.

Flea will eventually stuff that list of cylinders. Our current test mocks Flea and stuffs cylinders that draw a `T`. Our test must show `paintGL()` traversing that list, and drawing a line for each cylinder.

To add to that test assertions that fail without lines, copy in the line detection function calls from their previous test:

```
        static GLdouble const millimeters = 2000;
…
    TEST_(TestQt, addCylinder)
    {
        FleaOpenGL * pFOG = new MockQGLWidget;
        pFOG->eraseData();

                       //  thick     pig __begin__   _____end_____
                       //                x  y    z   x     y       z
        pFOG->addCylinder( 5131.25, 2, 0, 0,   0, 0,    0.00, 500.00 );
        pFOG->addCylinder( 265.625, 2, 0, 0, 500, 0,  176.77, 676.77 );
        pFOG->addCylinder( 265.625, 2, 0, 0, 500, 0, -176.77, 676.77 );

        pFOG->initializeGL();
        pFOG->resizeGL(500, 400);
        pFOG->paintGL();

        QString log = fileToString("glTrace.txt");

        GLdouble const m = millimeters;

                          // y    z      x     y           z         x
        log = findLine( log, 0,   0 / m, 0,    0.00 / m, 500.00 / m, 0 );
        log = findLine( log, 0, 500 / m, 0,  176.77 / m, 676.77 / m, 0 );
        log = findLine( log, 0, 500 / m, 0, -176.77 / m, 676.77 / m, 0 );

      //  reveal();  //  wait for it!
    }
```

Notice *x*, *y*, *z* became *y*, *z*, *x*. Flea uses *z* for “up”, while OpenGL uses it for “towards the frustum”.

Predict the tests will crash on an `assert()` inside `findLine()`. Then upgrade the production code:

```
        void
    FleaOpenGL::positionCylinder(primitive & aCylinder)
    {

        GLdouble x1 = aCylinder.Xbegin / millimeters;
        GLdouble y1 = aCylinder.Ybegin / millimeters;
        GLdouble z1 = aCylinder.Zbegin / millimeters;

        GLdouble x2 = aCylinder.Xend / millimeters;
        GLdouble y2 = aCylinder.Yend / millimeters;
        GLdouble z2 = aCylinder.Zend / millimeters;

        glLineWidth(aCylinder.thickness / millimeters * 250);

        gl->glBegin(GL_LINES);
          gl->glVertex3d( y1, z1, x1 );
          gl->glVertex3d( y2, z2, x2 );
```

```
        gl->glEnd();

    }

        void
    FleaOpenGL::paintGL()
    {
        prepareToPaint();

        gl->glMatrixMode(GL_MODELVIEW);
        gl->glLoadIdentity();
        gl->glTranslated(0, 0, -10.0);

        glEnable(GL_LINE_SMOOTH);
        glColor3ub(0,0,0);

        primitives_t::iterator it = cylinders.begin();

        for(;  it != cylinders.end();  ++it)
            {
            positionCylinder(*it);
            }

    }
```

The stunning, breathtaking result:

To repeat our performance, for spheres, let's add lines to the current test. We will put a sphere at the end of the left arm of that Y, and another, bigger one floating over the top. Ideally, you would clone the current test and convert it to one that tests only spheres:

```
    TEST_(TestQt, addCylinder sAndSpheres)
    {
…
                    //  diameter     pig   x      y        z
        pFOG->addSphere( 265.625,   4,   0, -176.77,  676.77 );
        pFOG->addSphere( 500.000,   4,   0,    0.00, 1000.00 );
…
    }
```

Those lines generate this new infrastructure:

```
        class
    FleaOpenGL:  public QGLWidget
    {
      public:
…
        void
    addSphere
            (
            GLdouble diameter,
            int      pigment,
            GLdouble X,
            GLdouble Y,
            GLdouble Z
            )
```

```
        {
        primitive aSphere = { diameter, pigment, X, Y, Z };
        spheres.push_back(aSphere);
        }

        void
    eraseData()
        {
        cylinders.clear();
          spheres.clear();
        }

      private:

        struct primitive {
            GLdouble  thickness;
            int       pigment;
            GLdouble  Xbegin, Ybegin, Zbegin;
            GLdouble  Xend,   Yend,   Zend;
            };

        typedef std::list<primitive> primitives_t;
        primitives_t cylinders;
        primitives_t spheres;

        void positionCylinder(primitive & aCylinder);
        void positionSphere(primitive & aSphere);

    };
```

To force spheres to appear in our output, go to `TEST_(TestQt, icosahedron)`, and perform Extract Method Refactor on the code that detects icosahedra to produce this new test fixture:

```
        struct
    TestQt: virtual TestCase
    {

        QString
    detectIcosahedron(QString logString)
        {
          //  parse out the GL_TRIANGLES

        QRegExp triangles("glBegin\\(GL_TRIANGLES\\)(.*)glEnd");
        triangles.setMinimal(true);
        int found = triangles.match(logString);
        assert(0 < found);
        QString z = triangles.cap(1);

        GLdouble haPhi = PHI / 2;

        z = skip(z, "glNormal3d");
        z = check3d(z, "glVertex3d", haPhi, 0,  0.5);
…
        uint index = triangles.pos(1) + triangles.cap(1).length();
        return logString.mid(index);
        }
…
    };
```

Our test calls that to count icosahedra:

```
        log = findLine( log, 0, 500 / m, 0, -176.77 / m, 676.77 / m, 0 );

        log = detectIcosahedron(log);
        log = detectIcosahedron(log);
```

Author the system to position them correctly:

```
        void
    FleaOpenGL::paintGL()
    {
...
        glColor3d(0.9,0.9,0.9);

        it = spheres.begin();

        for(;  it != spheres.end();  ++it)
            {
            positionSphere(*it);
            }
    }

        void
    FleaOpenGL::positionSphere(primitive & aSphere)
    {
        gl->glPushMatrix();
          gl->glMatrixMode(GL_MODELVIEW);

          gl->glTranslated
                (
                aSphere.Ybegin / millimeters,
                aSphere.Zbegin / millimeters,
                aSphere.Xbegin / millimeters
                );

          GLdouble xyz = aSphere.thickness / millimeters / 2;
          gl->glScaled(xyz, xyz, xyz);

          drawSphere();

        gl->glPopMatrix();
    }
```

Notice that system repositioned spheres by temporarily changing the definition of reality. `glPushMatrix()` preserves the previous definition, `glTranslated()` moves the origin, and `glScaled()` changes the metric. This system permits us to reuse `drawSphere()` without infesting all its `glVertex()` calls with extra math.

Check the result:

As we near the end of this Case Study, note the test code has a little design debt. Test fixtures like `check3d()` and `findIcosahedron()` repeat their common behavior to return the remainder of a `logString`.

Fold duplication when you have a quorum of these aspects:

Fold duplication that displays most of these aspects:

1. you feel inspired
2. the language permits
3. the duplicated statements express behavior, not structure
4. the code is production, not test code
5. the result won't obfuscate things
6. new abstractions might lead to the Open Closed Principle.

We lack items 1, 4, 5, and 6. Peeking ahead at the requirements, we won't implement any more Flea primitives during this iteration, so we need no more low-level OpenGL test fixtures. Folding that duplication would be too easy to prolong our next phase, and reaping the rewards of all this labor.

Editing Fractals

The whiteboard test plan, on page 321, is now within reach:

```
TEST_(TestQt, Tee)
{
    setSource( "egg = startOvum(1, 90, 100)\n"
               "tubes = egg.newRootAxiom()\n"
               "\n"
               "tubes.longer(5).\n"
               "    tube.tube.shorter.\n"
               "    push.right.tube.pop.\n"
               "    left.tube.\n"
               "    sphere\n"
               "\n"
               "render(egg)" );

    FleaOpenGL * pFOG = new MockQGLWidget;
    pMainWindow->evaluate();

    pFOG->initializeGL();
    pFOG->resizeGL(500, 400);
    pFOG->paintGL();

    QString logString = fileToString("glTrace.txt");

    logString = findAnyLine(logString);
    logString = findAnyLine(logString);
    logString = findAnyLine(logString);
    logString = findAnyLine(logString);
    detectIcosahedron(logString);

    reveal();

}
```

That looks much less impossible, now. Clone and modify the fixture `findLine()` to create `findAnyLine()`, and remove its assertions that constrain a line's location.

The remaining work requires understanding more about Flea's architecture, so we will start with its … interesting System Metaphor.

- `egg = startOvum(1, 90, 100)`—Flea grows fractals from `egg`s, with these settings:
  - `1`—recursion depth before retiring the turtle
  - `90`—default angle for turtle course corrections
  - `100`—default `tube` length.

- `tubes = egg.newRootAxiom()`—each `egg` contains `Axiom`s, which are lists of instructions to the turtle. Flea inherits the term "Axiom" from previous Lindenmayer Systems fractal engines. "Chromosome" would be better.

- `tubes.longer(5).`—The first action makes tubes longer. Each action returns its Axiom, so these action statements can chain together. The dot `.` links the next lines.
  - `tube.tube.shorter.`—draw two tubes, then make subsequent tubes shorter.
  - `push.right.tube.pop.`—preserve the turtle's current location, yaw 90 degrees to the right, draw a tube, and restore the preserved location and orientation.
  - `left.tube.`—yaw 90 degrees to the left, draw a tube.
  - `sphere`—add a sphere to the end of one tube, and retire.

- `hatch(egg)`—Unpack all those instructions, construct an initial Turtle object, and apply each instruction to it, outputting primitive instructions to the backend renderer.

An industrial-strength project should not mix System Metaphors and permit confusion! The turtle does not come out of the egg—the list of instructions to the turtle do. Flea inherits jargon from legacy Lindemayer Systems that use similar (yet mixed) metaphors.

Copy these Flea files into FleaGL's home folder:

- `fleaCore.rb`
- `trace.rb`
- `fleaGeometry.rb`
- `fleaConfig.rb`

Flea needs glue functions, for Ruby to find our C++ code. That new test will indirectly constrain all this code:

```
    static VALUE
addFleaGLCylinder
        (
        VALUE self,
        VALUE thickness,
        VALUE pigment,
        VALUE Xbegin,
        VALUE Ybegin,
        VALUE Zbegin,
        VALUE Xend,
        VALUE Yend,
        VALUE Zend
        )
{

    pMainWindow->glWidget->addCylinder
```

```
        (
        NUM2DBL(thickness),
        NUM2INT(pigment),
        NUM2DBL(Xbegin),
        NUM2DBL(Ybegin),
        NUM2DBL(Zbegin),
        NUM2DBL(Xend),
        NUM2DBL(Yend),
        NUM2DBL(Zend)
        );
    return Qnil;
}

    static VALUE
addFleaGLSphere
        (
        VALUE self,
        VALUE diameter,
        VALUE pigment,
        VALUE Xbegin,
        VALUE Ybegin,
        VALUE Zbegin
        )
{

    pMainWindow->glWidget->addSphere
        (
        NUM2DBL(diameter),
        NUM2INT(pigment),
        NUM2DBL(Xbegin),
        NUM2DBL(Ybegin),
        NUM2DBL(Zbegin)
        );
    return Qnil;
}
```

Upgrade `MainFleaWindow::evaluate()`, to bind those glue functions, and to read a new Ruby source file:

```
    void
MainFleaWindow::evaluate()
{
    QString source = sourceEditor->text();
    traceEditor->setText("");

    ruby_init();
    ruby_script("embedded");
    rb_define_global_function("sink", RUBY_METHOD_FUNC(sink), 1);

    rb_define_global_function( "addFleaGLCylinder",
              RUBY_METHOD_FUNC(addFleaGLCylinder), 8 );

    rb_define_global_function( "addFleaGLSphere",
              RUBY_METHOD_FUNC(addFleaGLSphere), 5 );

    source = "class MockStream\n"
             "    def write(x)\n"
             "        sink(x)\n"
             "    end\n"
             "end\n"
             "$stdout = MockStream.new()\n"
             + fileToString("fleaOpenGL.rb")
```

```
            + source;

    glWidget->eraseData();
    int state = 0;
    rb_eval_string_protect(source, &state);

    if (state)
        {
        VALUE c = rb_funcall(rb_gv_get("$!"), rb_intern("to_s"), 0);
        traceEditor->insert("ERROR ");
        traceEditor->insert(RSTRING(c)->ptr);
        traceEditor->insert("\n");
        }

    glWidget->repaint();
}
```

The final bit of glue is the new file "`fleaOpenGL.rb`". It satisfies the `Backend` interface:

```
# bond Flea with "FleaGL.cpp"

  #  add current folder to library path
$LOAD_PATH.unshift('.')

require 'fleaCore.rb'

Ovum = Flea::Ovum

class OpenGLBackend
    def addCylinder( length, thickness, pigment,
                     zeeRotation, whyRotation,
                     xyzTranslation, xyzEnd )

        addFleaGLCylinder( thickness, pigment,
                           xyzTranslation[0],
                           xyzTranslation[1],
                           xyzTranslation[2],
                           xyzEnd[0],
                           xyzEnd[1],
                           xyzEnd[2] )
    end

    def addSphere( diameter, pigment, xyzTranslation )

        addFleaGLSphere( diameter, pigment,
                         xyzTranslation[0],
                         xyzTranslation[1],
                         xyzTranslation[2] )
    end

    def addTriangle(xyz1, xyz2, xyz3, pigment)
        addCylinder(nil, 1, 1, nil, nil, xyz1, xyz2)
        addCylinder(nil, 1, 1, nil, nil, xyz2, xyz3)
        addCylinder(nil, 1, 1, nil, nil, xyz3, xyz1)
    end  #  TODO  add a "polygon" capacity to FleaGL

end

def render(egg)
    out = OpenGLBackend.new()
    egg.makePovRayScript(out)  #  TODO  needs a better name...
```

```
end
```

Now `TEST_(TestQt, Tee)` can finally pass. The visual appearance reveals our usability scenario. Users author Flea scripts in the left panel, and see OpenGL's results on the right:

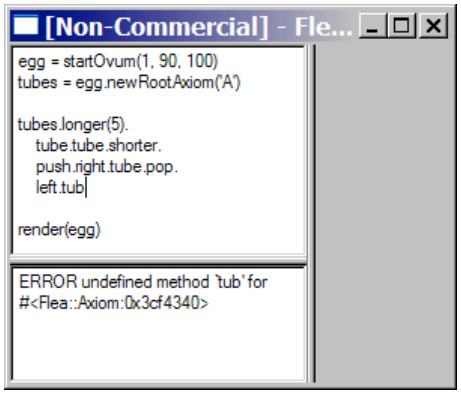
To enable that usability, upgrade `MainFleaWindow::MainFleaWindow()` to bind <F5> to `MainFleaWindow::evaluate()`:

```
#include <qaction.h>
...
MainFleaWindow::MainFleaWindow() :
            sourceEditor(NULL),
             traceEditor(NULL)
{
...
    QAction * theTestButton = new QAction(this, "theTestButton", TRUE);
    theTestButton->setAccel(Qt::Key_F5);

    connect( theTestButton, SIGNAL( activated() ),
             this, SLOT( evaluate() ) );

}
```

Within a primitive usability envelope, this project is ready to draw fractals. In theory, one could compile in Release Mode and run this program like a normal Qt application. I will just leave `reveal()` turned on while editing fractals:

```
pFOG->gl = &ogl;  //  so animation won't be slow
reveal();
```

We start with a *Temporary Interactive Test* of a mistake, to ensure we can recover. I erased the `e` on the end of `tube`:



The output panel correctly revealed the error: `undefined method 'tub'`.

Now grab a fractal from `http://flea.sourceforge.net/reference.html`, paste its source in, and fix its `depth` variable:

Notice that if `depth` is too high, the turtle spends too long walking the entire perimeter of that proto-fractal, and performance degrades. Flea uses sketching backends for rapid, low-fidelity feedback, before sending a fractal to a slow, detailed renderer. Set `depth` low when sketching, and high when rendering.

Those commands create a `root` axiom, whose only job is to make the turtle `longer`. Then `gosub(ax)` links that `root` to another axiom, `ax`, who does all the work. `ax` gets a little shorter, draws a `tube`, and pushes the turtle's current situation onto a stack. Then `ax` instructs the turtle to turn `right 60` degrees (based on `startOvum(depth, 60, 80)`), and `link` to `ax`. `link()` works like `gosub()`, except it counts its own recursion depth. When this exceeds `depth`, `link()` stops working.

The turtle recursively draws bent branches, until it returns from `link`. Then it `pops` its situation off the stack, and draws a `tube`. Then it uses `push` and `pop` to draw another branch to the `right`, and finally turns `left` & `link`s to itself again.

Edit Flea scripts by making small changes and banging <F5>. (Does that sound familiar?)

These changes make our tree look less flat:

```
depth = 6
egg = startOvum (depth, 60, 80)
root = egg.newRootAxiom('Q')
ax = egg.newAxiom('A')
root.longer(12).gosub(ax)

ax.shorter(0.5).tube.
   push.
      right.rollLeft(66).link(ax).
   pop.
   tube.
   push.
      right.rollRight(66).link(ax).
   pop.
left.rollLeft(66).link(ax)

render(egg)
```

A monochrome stick figure of a tree, in winter, belies OpenGL's prowess—and Flea's! We will add one more feature, to help FleaGL compete with the other Flea sketching interfaces.

Revolutionary Fractals

A slowly rotating image provides a compelling illusion of depth. Start by authoring a window timer:

```
    #include <qtimer.h>
…
        class
    MainFleaWindow: public QMainWindow
    {
        Q_OBJECT

    public slots:
        void evaluate();
        void timeout();
…
    };
```

```
...
MainFleaWindow::MainFleaWindow() :
            sourceEditor(NULL),
             traceEditor(NULL)
{
...
    QTimer *timer = new QTimer( this );
    connect( timer, SIGNAL(timeout()), SLOT(timeout()) );
    timer->start( 100 );  //  1/10th of a second

}

    void
MainFleaWindow::timeout()
{
    glWidget->repaint();
}
```

Now `FleaOpenGL` gets a new data member, to record the current degree of spin:

```
    class
FleaOpenGL:  public QGLWidget
{
...
  private:
    int degrees;
...
};

FleaOpenGL::FleaOpenGL(QWidget * parent):
        QGLWidget(parent),
        gl(&ogl),
        degrees(0)
   {}
```

Now `paintGL()` rotates reality before positioning the primitives:

```
    void
FleaOpenGL::paintGL()
{
    prepareToPaint();

    gl->glMatrixMode(GL_MODELVIEW);
    gl->glLoadIdentity();
    gl->glTranslated(0, 0, -10.0);

    gl->glRotated(degrees += 5, 0, 1, 0);

    glEnable(GL_LINE_SMOOTH);
...
}
```

As we author fractals, they slowly rotate:

This tiny program is now useful. As you edit a shape, each time you tap <F5>, you get instant, predictable feedback. A rotating silhouette sounds useless. In practice, it provides a complete illusion of shape. Your eye tracks features, and your neurons automatically rate their relationships to other features, without guessing.

If these fractals had some business value (beyond distracting workers from playing DarkWatch), they would be natural candidates for *Broadband Feedback*. Adding the required features to our existing test fixtures would be trivial.

Conclusion

Imagine life with a demanding colleague who cannot (or should not) write programs using hard languages. They keep asking you for trees, then for minor tweaks on them. Leaves must be greener, branches must be thicker, trunks must be thinner, etc. Your quality of life might degrade.

Agile software development thrives on such colleagues. We seek and remove duplication from source design, and from team behavior. The more often you re-author some detail, the more mandate you have to automate that authoring and return control to the whole team.

Flea is a declarative language. Any traditional LSystem fractal can express in Flea as a series of commands: `push.right.link(ax).pop.tube`, etc. This provides the convenience of programming without the cognitive mismatch of a procedural paradigm.

Just as teams alternate between fuzzy whiteboards and distinct source code, programs alternate soft and hard layers to split modules that require careful tests from those that only require authoring. Our Flea code emits only two kinds of errors. Syntax errors appear in our trace window, and semantic errors appear as ugly OpenGL animations. Scripting languages decouple modules to make changes easy and safe.

For our final Case Study, we research a system invented to divorce authoring from logic, which then spent a decade endeavoring to reunite them.

Chapter 12: *The Web*

Regardless of which GUI Toolkit is most popular, when tests drive GUI development, they should treat its Toolkit as *Just Another Library*. The word "Library" is singular.

When your GUI library pops up windows, interrupting Flow, you have problems.

When your GUI "library" is really a whole bunch of variously implemented, variously compatible protocols wrapped around a client-server model running a plethora of independently invented and exciting but prematurely delivered features, you have the World Wide Web.

Humans respond to minor problems that grow popular, like the web, by repressing symptoms, generating new problems, selling fixes to these, and generally creating major problems.

For example, the web informally prefixes its HTTP servers with "`www.`" This prefix broke the style guideline "use pronounceable variable names". Nobody reviewed that prefix with such a principle in mind. After the web became popular, those charged with pronouncing "double-you double-you double-you dot" all day, such as radio personalities, suffered. The prefix could have been "`web.`" It still could be. My favorite escape is "dub-dub-dub-dot", but I digress.

This Case Study's top half surveys test-first techniques to drive web development. The bottom half upgrades a Wiki into an Acceptance Test Framework, to present a gallery of images from *Broadband Feedback*.

All other Case Studies grow a project from scratch (or a previous Case Study). This Case Study adds a complex feature to a pre-existing, complex project. The reader is advised *not* to "play along" with these code samples. Get your own Wiki, and add these features to it.

Readers should adapt the spirits these techniques to their own Web projects.

HTML in a Nutshell

For those of you who lived during the past two decades under a very large rock, HyperText Markup Language is a language to mark hypertext up. HTML source mixes raw text with `<tags>`. A browser typesets the text based on the tags' intent, not exact typographical

specifications. The tags `<strong><em>text</em></strong>` could display bold italic ***text***, or could pronounce "text" loud and long.

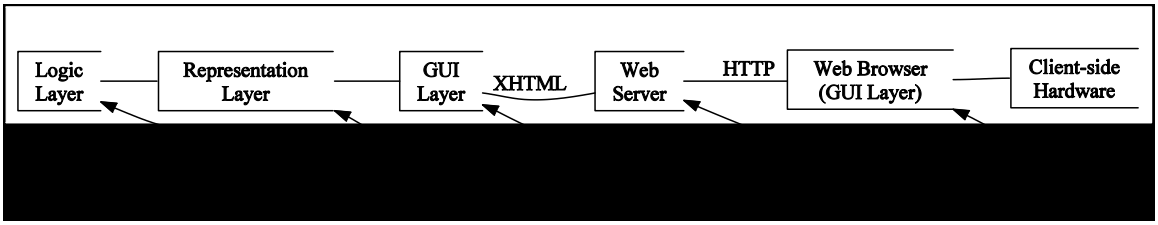
HTML pages link to network resources, including other HTML pages, using an anchor tag: `<a href="http://someserver/somepage.html">click me</a>`. The hypertext reference can use a wide variety of protocols to point to a wide variety of content types. When the target is another web page, HTML adds a new dimension to a linear reading experience.

HTML also serves as a remote user interface for server applications. Users enter data into web pages through the tags `<form>` and `<input>`. They write text into `<input type="edit">` fields, and click on an `<input type="submit">` button. The browser sends all `<input>` data to the server specified in the `<form>` tag. That server pushes the data into a script, to process the data, and returns a new page. The most common system to bind servers and scripts is Common Gateway Interface (CGI).

HTTP (HyperText Transfer Protocol) strongly decouples browsers from servers. The two sides are worlds apart. All browsers obey some subset of a wild assortment of overlapping HTML variations. Most HTML browsers speak Java, JavaScript (no relation), and Cascading Style Sheet (CSS). Commands to browsers must limit their platform-specific assumptions. Servers, by contrast, are free to process their side of the HTTP pipe using any combination of platforms, languages, programs, and databases. Tests constrain this riot of systems and protocols by following the same decoupling, and by restricting HTML to pure XHTML.

Test Tiers

Here's a Web application's architecture and test options. Our "GUI Layer" splits in two. Most of it ought to execute on the server side of the HTTP link. Some of it (DHTML, JavaScript, CSS, Java Applets, etc.) executes on the client side:



Each layer's tests generally mock the layer to its right. The Representation Layer thinks its tests are the server-side GUI Layer. The GUI Layer creates XHTML, thinking a Web server will transmit it. The Web server broadcasts HTTP to clients that it thinks are Web browsers. And Web browsers, hosting the client side of the GUI Layer, create "Document Object Model" representations with hopes that they will paint into display hardware, and users will look at them.

At each layer boundary, tests intercept I/O by mocking its protocol. XHTML Tests sample XHTML directly from the functions that create it. HTTP Tests emulate your browser, and stream the XHTML out of an HTTP server. DOM Tests enslave a real Web browser, and command it to navigate to your sample pages. Then test manipulate the objects that DHTML and JavaScript use.

To test-first a Web project, spend a little up-front design effort pushing all hard logic down below the Representation Layer. Most of that code will run headless, permitting the GUI Layer to shrink down as thin as possible. If they are so thin they only need Authoring, you are almost done. Write a few XHTML Tests, HTTP Tests, or DOM Tests, just to prove you can, and to

kick-start those modules. Most web site development consists of authoring and reviewing content.

The more dynamic your client-side appearance and behavior, the more tests you need to the right. To illustrate simple solutions for such hard problems, this book will test some easy things in hard ways. Sometimes we will use DOM Tests where XHTML Tests are indicated.

Minimize System Diversity

Web apps require a mix of programming languages, scripts, systems, and platforms. Make sure a development workstation can hold an image or miniature version of the entire project, from database to browser. This enables programmers to experiment with any part.

Write most tests in the language most convenient for the testee. Preferably the same language, but sometimes that's impossible. When you edit, and hit the One Test Button, the module-level tests evaluate in the common language. But, at need, test cases may shell to command-lines that run more tests in a different language.

For example, consider XSLT (explored on page 392) that converts XML into XHTML. You could write tests in XSLT, but if most of your logic is in Java then you will have trouble changing your mindset, and the test fixtures, between the two languages. So, write tests in Java language, and use an XSLT interpreter to produce the XHTML. Then test the XHTML using XPath for Parsed Fuzzy Matches, to demonstrate that the XSLT produced the correct fields. XPath queries can find target nodes insensitive to their surrounding details.

XHTML Tests

Most web tests need only assert that lower layers generate correct HTML with useful contents. The book *Extreme Programming for Web Projects*, by Doug Wallace, Isobel Raggett, and Joel Aufgang, provides very good advice to resolve duplication between the many modules that form a web application. Their Prime Directive: Convert all data into XML, author all HTML markup inside XSLT, and transform them together at the last moment to produce each page.

Not all HTML projects are content-rich data-driven public web sites. *TFUI* can only recommend that all output be XHTML, regardless of its source. *WebXP* complies with *TFUI* when the XSLT enforces XHTML output:

```
<xsl:output method="xml" media-type="text/html"
     standalone="no" omit-xml-declaration="yes"
     encoding="UTF-8"/>
```

XHTML Tests show XPath seeking expected values inside XHTML. The fun starts on page 378.

WebXP's Prime Directive, XML for data and XSLT for markup, provides a lot of incidental testing at very low cost. The act of parsing also tests. The book also describes a healthy lifecycle for incrementally adding and reviewing a site's esthetic content (its verbiage and graphics). "At all times a complete working site can be browsed in entirety without errors." Does that sound familiar?

Bootstrapping CGI Tests in Perl

Each TFUI development effort begins in a tricky spot. Your project must learn to use test-first, and not your platform's wizard *or* its web server, typically without support from your GUI Toolkit's documentation.

This Case Study will eventually add features to an existing project, and bypass CGI techniques. HTML tests are efficient, and "Common Gateway Interface" is the most common web server scripting layer, so we must bootstrap a new sample project, to see how it's done.

To add languages to this book's back cover, we bootstrap in Perl. This script squeezes two modules together—a `GuiLayer`, and its test, called `main`:

```
#!/usr/bin/env perl -w

use strict;  #  don't leave home without it
use CGI;
use Test::Unit::TestRunner;
use XML::XPath;

package GuiLayer;

    sub processRequest
    {
        my $self = shift;
        my $cgi = shift;

        my $payload = $cgi->param("payload");

        return $cgi->html(
                    $cgi->body("This page says $payload")
                    );
    }

package main;
    use base qw(Test::Unit::TestCase);

    sub test_verify_page_contents {
        my $self = shift;

    # make CGI think a web browser called it to build a page

        $ENV{'REQUEST_METHOD'} = 'GET';
        $ENV{'QUERY_STRING'} = 'payload=Daddy+Warbucks';
        my $cgi = CGI->new();

    # build the page

        my $response = GuiLayer->processRequest($cgi);

    #  parse the page's contents

        my $xp      = XML::XPath->new(xml => $response);
        my $payload = $xp->find('/html/body/text()');

    #  check our payload influenced the page

        $self->assert_matches(qr/Daddy Warbucks/, $payload);
    }

my $test = Test::Unit::TestRunner->new;
```

```
$test->start('main');  #  Test Collector pattern
```

The test makes `$cgi` and `GuiLayer` think a web browser requested its page, obeying an URL such as:

```
http://localhost/cgi-bin/whatever.pl?payload=Daddy+Warbucks
```

CGI passes data between a web server and a script by packing it all into environmental variables. Perl's CGI library permits us to spoof these data using by passing a string or a map into its constructor: `CGI->new( "payload=Daddy+Warbucks" )`, or `CGI->new({ payload => 'Daddy Warbucks' })`. I pushed the variables directly into Perl's `@ENV` representation of the OS's environmental variables. That technique can port to any CGI platform.

If your `QUERY_STRING` must contain a more complex URL, remember to use `URLEncode()` to escape any non-ASCII parameters. I represented a space as its common URL replacement, `+`.

A normal CGI script would call `print GuiLayer->processRequest($cgi)`. Our test intercepts the XHTML return value, and parses it with our favorite utility, XPath. Then `assert_matches()` checks that it contains our sample string. That Regular Expression Match, per page 37, could have been arbitrarily complex.

The `GuiLayer` module depends on no web server, only `$cgi`, an object that's easy to mock. If `GuiLayer` used any lower modules, the ones containing business logic would naturally avoid `$cgi`.

(Thanks to Tony Byrne, on the TFP mailing list, for seeding this topic.)

Temporary Visual HTML Inspections

If our Perl test sample had been more complex, we might need to visually inspect the XHTML it produced. If `$response` contained that XHTML, these lines would lead to a `reveal()` fixture:

```
open(TEMP, '>c:/temp/temp.html');
print TEMP $response;
close(TEMP);
system('start c:/temp/temp.html');
```

"`start`" depends on your MS Windows CMD.EXE, so you may need some other system to view the page, such as "`konqueror`".

Your HTML may pull in graphics and scripts from other files. One Hristo Deshev, from the TFUI mailing list, reminds us a temporary HTML file might not be able to see them, so use the tag Adding the tag `<base href="http://someUrl.com/" />` to your page will base all the relative path calculations on the URL.

HTTP Tests

A package that emulates a web browser, following the W3C recommendations, can keep all the real browsers honest. The Java library HttpUnit, by Russell Gold, at `http://www.httpunit.org/`, simulates a browser so well that one can develop server-side features using test-first on the client side. Don't make that a habit.

Temporary Visual HttpUnit Inspections

Many engineers' sanity depends on reading the daily comic strip "Dilbert". It illustrates the lives of highly intelligent engineers working for managers who owe their position to qualities other than intelligence.

When we software engineers read Dilbert on its web site, we should not endure all the popup adds and jiggling baloney around the payload. We need to test that Dilbert, in isolation from its web page, is funny.

This HttpUnit snippet reads the HTML page—without reading all its extras. Then it locates the actual Dilbert cartoon, downloads it to your `C:\temp` folder, and optionally presents it in Internet Explorer.

The presentation system can easily reconfigure for other browsers.

Note the `// reveal(page)` statement. If you de-comment it, you will see the Dilbert HTML page, without all its supporting images.

```
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.net.MalformedURLException;
import java.util.regex.Pattern;
import junit.framework.TestCase;
import org.xml.sax.SAXException;
import com.meterware.httpunit.*;

public class filchDilbert
  extends TestCase
{
    protected void setUp() {
        HttpUnitOptions.setScriptingEnabled(false);
//      HttpUnitOptions.setMatchesIgnoreCase(true);
    //  HttpUnitOptions.setImagesTreatedAsAltText(true);
//      HttpUnitOptions.setParserWarningsEnabled(true);
    }

    private void assertMatch(String regex, String input)
                                              throws Exception
    {
        boolean shouldMatch = Pattern.matches(regex, input);
        assertTrue("<" + regex + "> did not match <" + input +
                                            ">", shouldMatch);
    }

    public void reveal(WebResponse page)
        throws FileNotFoundException,
               IOException
    {

        String location = "c:/temp/test.html";
        FileWriter out = new FileWriter(location);
        String htmlContents = page.getText();
        out.write(htmlContents);
        out.close();
        explore(location);
        System.out.println(htmlContents);
    }

    private void explore(String location) throws IOException
```

```
        {
            Runtime.getRuntime().exec(
"\"C:\\Program Files\\Internet Explorer\\iexplore.exe\" " +
                        location);
        }
        private void saveBinary(File fileOut, String dilbertsGif)
                         throws FileNotFoundException, IOException
        {
            FileOutputStream fos = new FileOutputStream(fileOut);
            DataOutputStream stream = new DataOutputStream(fos);
            stream.writeBytes(dilbertsGif);
            stream.close();
            fos.close();
        }

        public void test_hitFrontPage()
              throws MalformedURLException, IOException,
                     SAXException, Exception
        {
            WebConversation     conversation = new WebConversation();
            String server = "http://www.dilbert.com/";

            WebRequest  request = new GetMethodWebRequest( server );
            WebResponse page = conversation.getResponse( request );

            assertMatch("Dilbert.*", page.getTitle());

    //      reveal(page);

    // find: <IMG SRC=
// "/comics/dilbert/archive/images/dilbert2091507040420.gif"
// BORDER=0 ALT="Today's Dilbert Comic">

            WebImage todaysDilbert =
                page.getImageWithAltText("Today's Dilbert Comic");

            System.out.println(todaysDilbert.getSource());
            request = todaysDilbert.getRequest();

            WebResponse response = conversation.getResponse(request);
            File fileOut = new File("C:/temp/dilbert.gif" );
            saveBinary(fileOut, response.getText());
            explore(fileOut.getAbsolutePath());
        }

    }
```

When HttpUnit helps grow your web site, you can leverage a `reveal()` fixture that takes a `WebResponse` argument. It writes the response's HTML contents to a temporary file, "`test.html`", and then raises that file in your web browser. Repeatedly displaying this page is much more efficient than repeatedly restarting your server and then surfing to the target page. Programmers use tests to force down the cost of the feedback required to write tests.

If your web page requires extra files, such as style sheets or images, your test fixture must ensure they are available in a folder where your Web browser can find them with the `file:` protocol. This implies the local folder keeps them in the same place, relative to your "`test.html`" file's location, as the server keeps them relative to the target page.

Insecurity

One more note about tests that spoof a user. If you write a public web site, and need security from automated scripts that simulate users, you must enforce some constraints that break testing. You might restrict the number of hits permitted from a single address, require an e-mail address and a mail-back, or ultimately implement a "Reverse Turing Test":

That's also called a "CAPTCHA", for "Completely Automated Public Turing test to tell Computers and Humans Apart". Humans conduct manual "Turing Tests" to learn if Artificial Intelligence solutions can impersonate humans. Computers conduct automated "Reverse Turing Tests" to ensure remote users are not other computers. These tests present challenges that humans can easily solve but computers cannot yet solve.

All these pernicious issues arise from the native insecurity of raw TCP/IP packets, which we must assume came from whom they say they came from. Until computer scientists and network administrators sort these problems out, down in their Logic Layer, we must write irritating security systems that make our public web sites test-resistant and slightly user-hostile. (And we must endure endless floods of spam and malware…)

Tests must temporarily disable security to do their jobs. A test might send a private web server a signal on a port other than 80 to disable its security, and the server could trust that any signals on ports other than 80 came from the local side of a firewall. This strategy leaves the security system intact for manual testing.

These security issues apply to all tests downstream from an HTTP server.

DOM Tests

The original web browsers displayed motionless text and graphics. To provide active client-side special effects, based on user interactions, HTML embedded a light scripting layer called JavaScript. (The language's official name is "ECMAScript", but nobody calls it that.) HTML pages embed this language inside `<script>` tags, and inside certain node attributes. A web browser publishes a few identifiers, such as `document`, to its JavaScript layer. Each identifier contains members that navigate a Document Object Model, representing all the nodes in a document. Put another way, most HTML tags turn into data objects when they render.

Some web browsers publish this model externally, through a standard Object Request Broker. Some browsers publish a back-door into their JavaScript layer, so a program can command the browser to fetch a web page, then evaluate a string containing JavaScript. Tests may exploit these systems to drive DOM and inspect its behavior. These tests have the benefit of running a real Web browser, not an emulator. They have the drawback of running only one breed of web browser. All browsers are different, so projects that rely on dynamic client-side behaviors must somehow provide Abstract Tests that run on each browser. As usual, limiting client-side behaviors will also narrow the risk of bugs.

Temporary Interactive MS Internet Explorer Tests

This Ruby code drives MS Internet Explorer to surf to a web page on my local server, write on its form, and click its Save button:

```
ie = WIN32OLE.new('InternetExplorer.Application')
ie.navigate('http://localhost:8080/WikiTranscludeText')

while ie.busy
```

```
        sleep(0)
    end

    until ie.readyState == READYSTATE_COMPLETE
        sleep(0)
    end

# ie.Visible = true

    formNode = ie.document.forms("files/sample.txt")
    assert_not_nil(formNode)
    inputNode = formNode.namedItem("contents")
    assert_not_nil(inputNode)

    inputNode.setAttribute('value', 'What\'s up Doc?')

    saveButton = formNode.namedItem("Save")
    assert_not_nil(saveButton)
    saveButton.click()
```

Note the line `# ie.Visible = true`. To *Temporarily Visually Inspect* this test, you might simply command the controlled Internet Explorer object to paint itself on your screen. However, that line does not block. As usual, a GUI Toolkit provides a convenience that both assists and interferes with our TFUI Principles. After displaying your web page, this test continues to run, and the subsequent test statements will evaluate. If they command IE to do other things, you will see them happen in real-time.

The strict *Temporary Interactive Tests* definition (on page **Error! Bookmark not defined.**) says, "All testing shall block until you close the window." That permits a window to appear in a predictable state before you click on it. DOM tests would need a system to regulate IE's event queue, and block subsequent test statements until you closed IE.

The strict definition also says, "After the window closes, other tests shall run, if any." If you don't mind losing that Principle, you can fix both those issues terminally:

```
ie.Visible = true
exit(0)
```

The controlled Internet Explorer object can outlive the Ruby process that spawned it. Tests cease to drive DOM, and you can see your web page's exact condition at the place you chose to put those lines. The ability to predict a temporarily tested web page's state is more important than Incremental Testing, the practice that the strict *Temporary Interactive Test* definition defends.

Other Platforms

Because DOM enables scripting, any web browser could test through DOM. The trick is binding a script from your tests to the web browser's internal objects. Some browsers do not publish their DOMs on public ORBs. The "KDE Desktop Environment", for example, provides an exemplary web browser called Konqueror, and a JavaScript component called KJSEmbed, with a command-line interface called `kjscmd`. The link from external JavaScript to Konqueror uses a Qt system called "KParts", but this only publishes the browser's external methods, not the DOM inside it.

(Learn more about Qt on page 307.)

One drives that DOM by writing external JavaScript statements that build strings containing JavaScript, and submitting them to a Konqueror part through the external method `.executeScript()`, after the document loads.

(Notice the external script could use any language that supports KParts, not just JavaScript. The goal "Minimize System Diversity" can sometimes be a little specious.)

Thanks to Koos Vriezen for providing this bootstrapping example:

```
    #!/usr/bin/env kjscmd

    function Slots() {
        this.pageLoaded = function() {
             println ("loaded");

        part.executeScript(
'document.forms[0].elements("text").value="What\'s up Doc?"');
        }
    }
    var mw = new KMainWindow();
    var box = new QVBox( mw );
    var slots_obj = new Slots();
    mw.setCentralWidget(box);
    var part = Factory.createROPart("text/html", box, "html widget");
    part.openURL("http://127.0.0.1/wiki.rb?edit=WikiTranscludeText");
    mw.connect(part, "completed()", slots_obj, "pageLoaded");
    print(part.signals());
    mw.show();
    application.exec();
```

A test derived from that sample could return `0` for success or `1` for failure to its environment, and a test rig could shell to it and collect that value.

Note the statements `mw.show()` and `application.exec()`. Koos spot-checked his script using the *Temporary Interactive Test* Principle.

Test Integration

One button must test all aspects of every module, and one top-level `AllTests` command must test all modules. Our survey of techniques here challenges rambunctious web projects to minimize their language diversity. Ideally, only one language should host all tests. But our survey showed how different web layers can require tests in different languages.

Tests written in Ruby, for example, could shell to JavaScript (hosted in `ksjcmd`) to run a few DOM tests, and collect their return value to "bubble up" their status:

```
    result = system("kjscmd myDomTests.js")
    assert_equal(0, result)
```

The patterns of failures in these "shelled tests", and their interactions with your editors, will reveal when and how to improve their *Fault Navigation*.

When you change a test, get it to fail, change the source, and get it to pass, each test run must be as easy as a few keystrokes—hopefully just one. That's why you must spend a little extra effort to cobble together scripts and embedded command lines to put all these different test rigs, on both the server and client side, under one of your editor's buttons.

Ideally, a project that limits client-side JavaScript will require only a few DOM tests.

Before introducing our main project, I'm compelled to showcase a fine example of a test-first project that exploits advanced client-side JavaScript.

Mini Ruby Wiki

Unlike other Case Studies, this one cannot invite the reader to play along with the text, and make each change as presented. You have already learned the TFUI Principles for simpler projects; the project here must cover too much territory for each little refactor to provide readable prose.

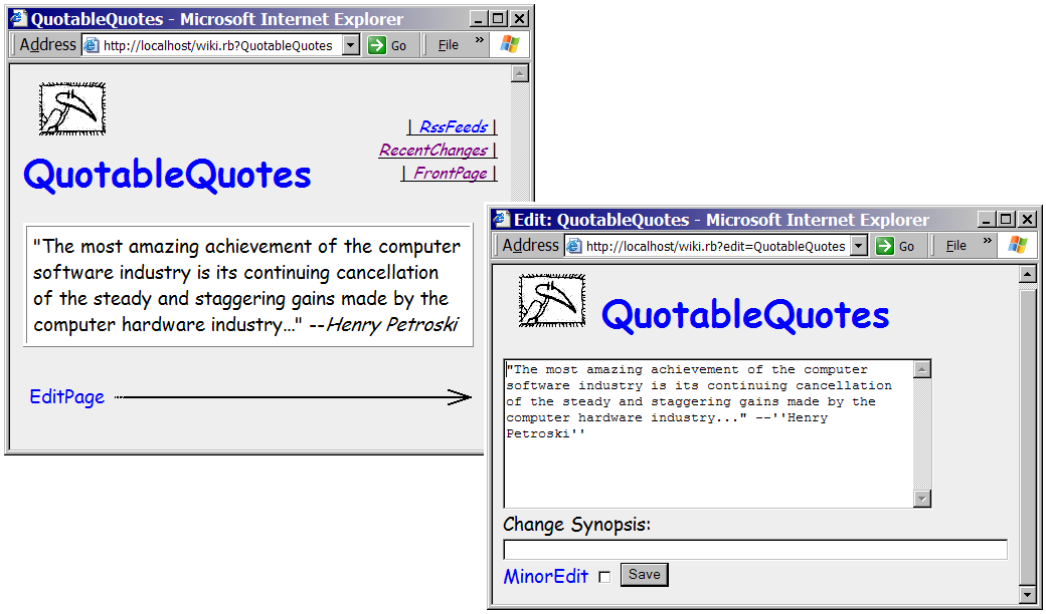
Wiki Wiki Web sites present a range of test challenges. If you never heard of a Wiki, stop reading now and e-search the Internet for "WikiWikiWeb". Ward Cunningham invented them, and co-wrote *The Wiki Way: Collaboration and Sharing on the Internet*, with Bo Leuf.

Some day there will be as many versions of Web sites with content that users can edit as there are Linux distributions. Their divergent features target various intents and whims. For example, my latest Wiki, called MiniRubyWiki (MRW), is written in Ruby, produces XHTML, and can display GraphViz output. (Try to survive those shocks!)

Wiki Wiki Webs

Humans, collaborating to write a Web site, most frequently want to upload simple paragraphs of text, containing a few markup codes and links to other pages in the same site.

The least simple solution gives all the users expensive "WYSIWYG" editors and complex upload protocols. The most simple solution gives each Web page an "EditPage" button. When you click it, you get a page containing the previous page's content area rendered as a big edit field. The content area contains not HTML source but a tiny breezy markup language:



Analyzing users' requirements, removing duplication from their behaviors, and trimming unneeded features all specify the simplest thing that could possibly work: A Wiki.

Duplication removal drives design. Source should strive to define program behavior Once and Only Once. Software should not require users to perform the same sequence of actions too often. So duplication removal drives requirements analysis.

Wiki Markup

Here are the two most common Wiki typesetting markups, using ticks `'`, with their outputs seen through default browser settings:

`this is ''em''phatic`—this is *em*phatic
`this is '''strong'''`—this is **strong**

All Wikis provide a few more complex markups. But Wiki notation obeys constraints different from other authoring systems. HTML strictly reserves the tag marker `<>`—nothing else may resemble one. But Wiki markup obeys naïve users' understanding, and advanced users' expedience, over strict markup purity. A single tick `'` is just an apostrophe. Wikis cannot indulge in arbitrarily complex markups. Many simply provide an "HTML Mode" for select pages, giving power-users free reign. And many reserve a special character at the start of a line, such as a bang `!`, to escape embedded commands.

External Links

Wikis express URLs, such as `news:comp.software.extreme-programming`, or `http://www.dilbert.com/`, as hypertext links to their targets. When such URLs begin with `http:` and end with a common graphics file extension, such as `.gif`, `.jpg`, or `.png`, Wikis instead display them using the `<img>` command. To decorate a Wiki with graphic content, put graphic files on a web server, and enter their addresses into your page.

Internal Links

The absolute genius of the original `www.c2.com` Wiki, honored in all its followers, is its internal link protocol. A word with leading and embedded capital letters, such as WardCunningham, renders a link to the nearest Wiki page with that name. Users create a growing KnowledgeBase full of project details, written on pages named using ProjectSpecificJargon.

Teams should deliberately create such jargon (their SystemMetaphor) and store it in their KnowledgeBase Wiki.

The reader indulges in the original meaning of "hyper" text—they can read a page from top to bottom, or they can read in the orthogonal dimension; out and back through other pages.

When Representation Layers Produce HTML

The core of every Wiki is a Representation Layer that converts Wiki markup into an HTML representation. MRW developed test-first. Its earliest features (with highest business value) transform one representation (the Wiki markup) into another (HTML).

These primordial assertions, for example, created and defended the Wiki markup:

```
assertEqLf(formatWiki("<br>"), "&lt;br&gt;\n")
assertEqLf(formatWiki("''br''"), "<em>br</em>\n")
assertEqLf(formatWiki("Yo"), "Yo\n")
assertEqLf(formatWiki("YoYo"), "YoYo<a href=\"edit=YoYo\">?</a>\n")
got = formatWiki('FrontPage')  #  a 'tag', or internal link
```

```
assert_match(/href="FrontPage/, got)

got = formatWiki('baBaLoo')    #   not a tag
assert_no_match(/href/, got)

got = formatWiki('WinThirty2')  #   not a tag either
assert_no_match(/href/, got)
```

(`assertEqLf()` Fuzzily Matches HTML, disregarding many blank characters.)

Bulk Parsed Fuzzy Matches

Because MRW produces XHTML, one can test-last entire pages using validating parsers.

Validating HTML parsers make respectable tests. We prefer XHTML because XPath can incidentally validate as it queries contents. Many other fine validators are available. When they leverage awareness of HTML structure, they can detect errors in the meaning of the markup tags. For example, one should not put a `<title>` tag inside a `<body>` tag.

The program Tidy, by Dave Raggett, can read HTML and return a list of these issues. Get it at `http://tidy.sourceforge.net/`.

MiniRubyWiki used Tidy, early in its development, to constrain its HTML output. The case `test_tidyDoesNotDespiseOurHTML()` converts each page in a list from Wiki format into HTML, passes this through Tidy, and parses the extracted error messages. For any non-trivial error, the test case prints out the offending HTML line, with a caret ^ pointing at the beginning of the offending codes.

This system does wonders to constrain refactors so they strictly preserve behavior. The user, and their browser, might not care if a given stretch of HTML said:

```
  <strong><em>so what?</strong></em>
```

or:

```
  <strong><em>so what?</em></strong>
```

Honestly, we still don't care either. But if a refactor accidentally inverted those two end tags, it could invert other things, too. Tidy will stop us as soon as possible, and print out the line that failed. We will (usually) ignore the error message, tap Undo until it goes away, and try again.

Hyperactive tests are good because they prevent sloppy refactors. A hyperactive test fails when a user would not have perceived a bug. TFP produces many such tests, because over-constraining is cheaper than exactly constraining. When such a test fails, only the most recent edit could be at fault. If a module leverages such tests, ensure they run with its tests, not with the integration tests.

Pouring an entire page into a validator might be called a "bulk assertion". Use such techniques aggressively, but watch out for two problems.

If we don't ignore the error message, it does not directly indicate which line of source code caused the problem. The validator will reveal the line of XHTML that offended it, not the line of your source that caused the error. The error messages, and your test case's diagnostic output statements around them, provide plenty of clues, but no smoking gun. Most of our tests approach the ability of a **Unit Test** to isolate our culprits.

Bulk assertions' biggest problem is they can't test-first. Validators that rely on HTML that already exists can't enforce new features.

Custom Controls

We will add to MRW some features more advanced than `<form>` and `<textarea>`. Diverse code, in several layers, support their behaviors, so each layer needs new kinds of tests. And some tests will target the wrong layer simply to prove we can write a test in a difficult situation.

Transclusion

"Transclude" means "transitive include". Our Wiki will go beyond passive inclusion of read-only material. We will add features to our Wiki that allow users to read and write formatted data, and execute programs with them.

Our Customer Team wants us to add new markup commands to Wiki page source that produce interactive controls. Some will provide client-side JavaScript, so we will soon need tests on that behavior. Those tests will need a fixture that drives Internet Explorer's DOM through its automation interface, so we will find an early excuse to write one.

The feature requests:

- Wiki pages can **Transclude** text files as `<textarea>` tags.
- A "**Save**" button shall save changes.
- Users can **clone** the text file to a new name.
- Cloned files appear **next** in the Wiki page.
- A "**Test**" button can save the text file, then send its name to a console program.
- The **output** of the console program appears, formatted, below the `<textarea>`.
- Wiki pages can transclude simple **XML**.
- The XML shall **render** as HTML.
- The HTML shall display **edit fields** for attributes, and a `<textarea>` for the content.
- Each node has a "**Clone**" button, which clones the node.
- Each node has a **Test** button, which sends the node's XPath and file name to a command.

All these activities—saving files, executing commands, etc.—run server-side. Don't expose MRW to the open Internet without further security precautions!

Transclude a Text File

The first decision: How to write a Wiki page that commands the Wiki format system to transclude a text file? Wikis display image files based on their extensions, but we will need more flexibility. Not all text files end in `.txt`.

Wikis often introduce their extension commands with a bang `!` in the first column. English grammar forbids exclamation points to start sentences, so this operator is free to "overload".

We might transclude a text file using only a bang and the file name:

```
!files/sample.txt
```

(We put the text files into a folder called "`files`", to keep them out of the Wiki's home folder.)

But peeking ahead at the requirements, we know we'll introduce more commands than just text transclusion, so we follow the bang with the command `text!`, then a test fixture `ruby -v!`, then a folder and file name:

```
!text!ruby -v!files/sample.txt
```

Those learning Agile techniques may have heard one never writes speculative code. It's true; the path from speculative code to code that adds value is always higher risk than the path from no code. However, if you peek ahead at requirements within this iteration, you might make a decision now that results in code just as simple as code written without peeking. You may speculate, just a little, so long as the current feature receives the same *volume* of behavior. If you guess wrong, the fix is always easy, because the speculation should not exceed the current iteration.

We'll develop the `<textarea>` by testing the Representation Layer:

```
def test_transcludeText()
    hello_world = "hello world"
    writeFile("files/sample.txt", hello_world)
    transcluder = "!text!ruby -v!files/sample.txt"

    contents = formatWikiTest(transcluder)
    assert_no_match(/#{transcluder}/, contents)

    xhtml = "<BODY>" + contents + "</BODY>"
    doc = Document.new(xhtml)
    textarea = XPath.first(doc, '//TEXTAREA')
    assert_equal(hello_world, textarea.text)
end
```

This small XHTML test writes a file, puts the filename into our new Wiki notation, and sends that to the middle-level MRW function `formatWiki()`. That returns an XHTML fragment, and the test examines it.

The first assertion…

```
    assert_no_match(/#{transcluder}/, contents)
```

…merely checks that we did not pass the raw "`!text!ruby -v!files/sample.txt`" through into the output page; that was the previous behavior.

The function `formatWiki()` converts Wiki notation into ill-formed XHTML, without a top-level tag. When the normal production functions call `formatWiki()`, they wrap more XHTML tags around it, so complete Wiki output pages are always well-formed.

In this test, to get `formatWiki()`'s output ready for XPath, we wrap the `contents` inside `<BODY></BODY>` tags. This is a Parsed Fuzzy Match:

```
    xhtml = "<BODY>" + contents + "</BODY>"
    doc = Document.new(xhtml)
    textarea = XPath.first(doc, '//TEXTAREA')
```

XPath mercifully ignores details we don't care about. It seeks a `<TEXTAREA>`, so our final assertion can see if this contains the sample text:

```
    assert_equal(hello_world, textarea.text)
```

The simplest way to make that test pass is intercept the bang, parse the command, read the file, and paste it into a `<TEXTAREA>`. Code deep inside `formatWiki()` soon does this:

```
…
            elsif line =~ /^!text!(.*)!(.*)/ then
                fileName = $2
                fileContents = readFile(fileName)
                line = '{' + fileName + '}<br/>'
                line += cgi.textarea{fileContents}
            else
…
```

MRW's working loop interprets each line in a Wiki page, delimited by `\n`. That new regular expression finds a bang at the beginning `^` of `line`. The regular expression parses out the file name; we read that file and build a text area.

I also authored the file name as a title for the text area. Note that `<br/>` is well-formed XML—it has the `/` end-of-tag marker in it. XPath, in the test, sees only pure XHTML.

This wasn't very hard because it does not address how the HTML performs after it appears in a browser. For the next feature, we must test user interaction, round-trip through the server:

- A "Save" button saves text changes.

We could test that with HttpUnit or WebUnit. It would open a client-side port to the HTTP server, then inspect the returned HTML. To get ready to test harder features, we will instead learn to test in the DOM layer.

DOM tests in Ruby

HTML specifies controls that something must render before we can test at the GUI Layer. These kinds of tests must use a web browser to complete the user experience, before we start testing that experience. The benefit is real objects, not strings, whose structure and arrangement are occasionally similar across platforms.

The market's leading browser, MS Internet Explorer, expresses DOM objects via Component Object Module Automation. Although any COM-enabled language can drive it, a public web project that needs DOM tests must also investigate its rivals (Konqueror, Lynx, Mozilla, Opera, etc.) to test their DOM layers too. Browser "forgiveness" bedevils web tests, and potential customers will not be overjoyed to see nothing but warnings, such as "This web site requires IE 7.5 or greater".

To learn to test in the DOM layer, we write a Learner Test that, for starters, does essentially the same thing as our last test. It will force us to add an element to the HTML; this time it will be the Save button. The test is long, and contains much that is not familiar yet, and much that is not hidden inside fixtures yet, so its discussion appears between its lines.

```
require('win32ole')
```

That wraps COM, turning most COM servers into Ruby objects. However, as we learn to invoke COM methods, we can't view them in lists using the same reflection channel as Ruby object methods. `.public_methods()` won't list them. This line provides a kind of "IntelliSense":

```
puts aNode.ole_methods.collect{|m| m.name }.sort()
```

Our goal is to click the Save button on a form, and ensure the server saved our file. The server runs in the same computer as our test, so we will see that file appear in the `files` folder.

Given `ie` holding an InternetExplorer.Application object, these lines extract a form, named after the text file it hosts, and extract the `<textarea>` from that form:

```
formNode = ie.document.forms("files/sample.txt")
assert_not_nil(formNode)
inputNode = formNode.namedItem("contents")
assert_not_nil(inputNode)
```

Because subsequent tests will often get input nodes out of forms, that code will soon become a fixture. Now we change the sample text by writing on the `<textarea>`:

```
inputNode.setAttribute('value', 'What\'s up Doc?')
```

Get a handle to the Save button:

```
saveButton = formNode.namedItem("Save")
assert_not_nil(saveButton)
```

From here, a simple `saveButton.click()` will simulate a user clicking that button. IE will send an HTTP POST message to the server, which should save the new contents to the file.

But these messages are asynchronous, so our tests must re-synchronize them by polling the `ie.readyState` property. After that indicates the page has refreshed, we will read the file and see if our new text appears in it.

All these test abilities are candidates for fixtures:

- Navigate IE to a test page
- Wait for IE to load a page
- Find our testee `<form>` object
- Find our testee `<textarea>`, and write on it
- Find our testee Save button, and click on it

After that test fails, we will add a trivial amount of code to MRW to make the test pass. But beating that tiny feature out of MRW required all this test code:

```
def test_saveTranscludedText()
    hello_world = "hello world"
    writeFile("files/sample.txt", hello_world)
    transcluder = "!text!ruby -v!files/sample.txt"
    writePage("WikiTranscludeText", transcluder)

    ie = WIN32OLE.new('InternetExplorer.Application')
    ie.Navigate('http://localhost:8080/WikiTranscludeText')

    while ie.busy
        sleep(0)
    end

    until ie.readyState == READYSTATE_COMPLETE
        sleep(0)
    end

    formNode = ie.document.forms("files/sample.txt")
    assert_not_nil(formNode)
    inputNode = formNode.namedItem("contents")
```

```
        assert_not_nil(inputNode)

        inputNode.setAttribute('value', 'What\'s up Doc?')

    #   ie.Visible = true  #   to inspect setAttribute worked
    #   exit(0)  #   prevents subsequent lines from changing IE

        saveButton = formNode.namedItem("Save")
        assert_not_nil(saveButton)

        saveButton.click()

        while ie.busy
            sleep(0)
        end

        until ie.readyState == READYSTATE_COMPLETE
            sleep(0)
        end

        nuSample = 'What\'s up Doc?'
        assert_equal(nuSample, readFile("files/sample.txt"))

    end
```

If you wrote a similar test using IE and your language, the sequence of COM calls to IE would be the same.

We labored this far to write tests that will force a code change which entry-level HTML producers could make, without tests, in their sleep. And entry-level TFP programmers could create these features by testing the Representation Layer directly, like all the other MRW tests do, in their sleep. So why is all this so hard? The documentation and tutorials for DOM libraries assume we are coding and fixing, not testing. Nobody documents how to drive IE just like this. But why are we doing it?

Remember the "Time vs. Effort" chart (page 53), at the end of "One Button Testing"? Where this feature is going, it will need all those kinds of tests to pin down exact behavior. But we are still on the up-slope of that effort curve. After we learn to drive DOM, we can reuse our test fixtures to test different things the same generic way.

The TFP literature generally recommends, "Don't write a whole new test only to create a Get or Set method." The TFUI literature recommends, "Don't write a whole new test just to force a button to exist." That's exactly what our tests on the DOM layer did, and tests on the HTML itself could have forced the new button to arrive faster.

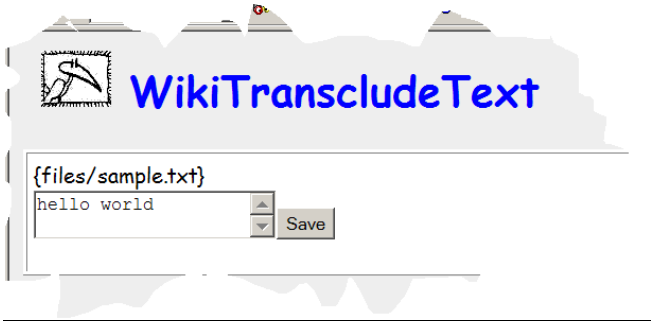
We picked a simple ability for our first test against DOM as a Learner Test. We are teaching ourselves (and our fixtures). We need to cross that "Time vs. Effort" divide where it's lowest. The tests that come will get harder, and we must be ready for them.

Oh, by the way, here's the code to pass the test. It uses CGI library methods to generate a `<textarea>` within a `<form>`:

```
    def insertTextArea(fileName)
        fileContents = readFile(fileName)
        line = '{' + fileName + '}<br/>'
        line += cgi.textarea('name'=>'contents'){fileContents}
        line += '<INPUT type="Submit" value="Save"/>'
```

```
        line += '<INPUT type="Hidden" name="returnTo" value="' +
                                    $currentPageTitle + '"/>'

        line += '<INPUT type="Hidden" name="fileName" value="' +
                                    fileName + '"/>'

        return cgi.form(method="POST", action="engaging"){line}
    end
```

Briefly, hidden tags now carry the source Wiki page name ("`returnTo`") and the source text file name ("`fileName`"). At POST time, MRW's CGI server reads the POST's `action` value, recognizes "`engaging`", reads the file name, and writes the file from the POST data called "`contents`". (That change is not shown—MRW's CGI layer is not exemplary. Your CGI documentation and tutorials most likely demonstrate these kinds of features.)



If you run MRW, view a page like that, change the text, and click the Save button, the new text saves to the named file.

Going forward, expect only 5% this much effort to set up a single test. Here's one of the fixtures that will make tests easier:

```
        def getNode(formID, inputID)
            formNode = @ie.document.forms(formID)
            return nil  if formNode.nil?
            inputNode = formNode.namedItem(inputID)
            return inputNode
        end
```

Tests in this suite will share a member variable, `@ie`, so we needn't pass the same one into each fixture.

Pull the next requirement:

- Users can **clone** the text file to a new name.

Our first version of the test only forces the clone button to exist. This lets us proof our new fixtures:

```
        def test_cloneTranscludedText()
            surfToSamplePage()

    #   locate the Clone button

            cloneButton = getNode(SampleFile, 'Clone')
```

```
        assert_not_nil(cloneButton)
        assert_equal('submit', getAttr(cloneButton, 'type'))
        assert_equal('Clone', getAttr(cloneButton, 'value'))
    end
```

That shows two new fixtures. The first, `surfToSamplePage()`, writes the sample Wiki page, and `Navigates` our member `@ie` to it, using code derived from our first DOM test. The second, `getAttr()`, needs a little explanation:

```
    def getAttr(aNode, attr_name)
        return aNode.attributes.
                getNamedItem(attr_name).value
    end
```

The "attributes" in a tag are the `<tag name="value">` things. DOM places them in the `attributes` property. We must learn to test attributes, no matter how deep in DOM they lurk, because not all are mere authoring, and not all appear in source explicitly. Schema systems like "Document Type Definition" files might quietly push in unexpected ones.

Now use the new fixtures to finish the test on the Clone feature:

```
def test_cloneTranscludedText()

# remove any leftover cloned text file

    file2 = "files/sample2.txt"
    File.delete(file2)  if File.exists?(file2)

    surfToSamplePage()

# locate the Clone button

    cloneButton = getNode(SampleFile, 'Clone')
    assert_not_nil(cloneButton)
    assert_equal('submit', getAttr(cloneButton, 'type'))
    assert_equal('Clone', getAttr(cloneButton, 'value'))

# locate the Clone button's new file name

    newFileName = getNode(SampleFile, "NewFileName")
    assert_not_nil(newFileName)
    assert_equal('text', getAttr(newFileName, 'type'))
    assert_equal(SampleFile, getAttr(newFileName, 'value'))

# change the file name

    newFileName.setAttribute('value', file2)

# after writing the above, set @ie.visible = true, and
# check IE shows the right file name

# click that Clone button, and check the new file showed up

    cloneButton.click()
    ieWait()
```

```
    assert(File.exist?(file2))
    assert_equal(Hello_World, readFile(file2))

    expect = "!text!ruby -v!" + SampleFile +
             "\n\n!text!ruby -v!files/sample2.txt"

    assert_equal(expect, readFile(wiki("WikiTestPage")))

end
```

The test case cleans up after its last run, starts its standard test page, changes the cloned file's name, and clicks the Clone button. (`ieWait()` spins until `@ie` finishes loading a page.)

Wiki authors will write this in a page:

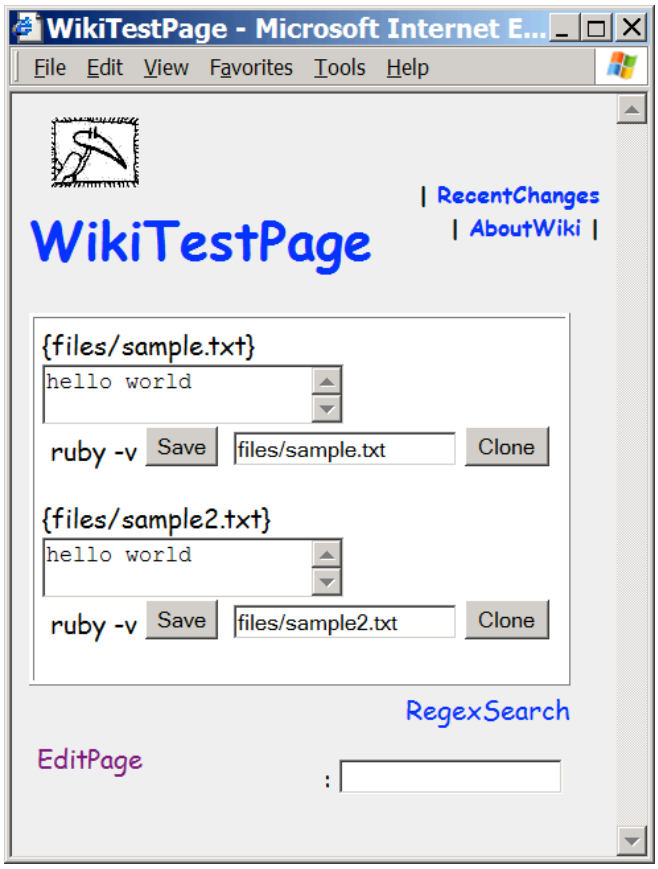
```
!text!ruby -v!files/myFile.txt
```

After cloning, the page should contain this…

```
!text!ruby -v!files/myFile.txt

!text!ruby -v!files/anotherNameFile.txt
```

…and "`files/anotherNameFile.txt`" receives the `<textarea>` contents.

After the test clones, the page appears like this:



The next test ensures if the contents of the `<textarea>` for `file/sample.txt` changed before cloning, `file/sample2.txt` would contain the new contents:

```
def test_cloneTranscludedTextWithNewContents()
    file2 = "files/sample2.txt"
    File.delete(file2)  if File.exists?(file2)

    surfToSamplePage()

    cloneButton = getNode(SampleFile, 'Clone')
    newFileName = getNode(SampleFile, "NewFileName")

    assert_equal( SampleFile,
                  getAttr(newFileName, 'value') )

    newFileName.setAttribute('value', file2)
    textArea = getNode(SampleFile, 'contents')
    nuSample = 'new sample'
    textArea.setAttribute('value', nuSample)

    cloneButton.click()
    ieWait()

    assert(File.exist?(file2))
    assert_equal(nuSample, readFile(file2))
    assert(Hello_World, readFile(SampleFile))
end
```

That test enforces the design goal that the first file does not also contain the new text contents, and the test already passes.

Least Favorite Editor

A note about my environment. Per the advice from Andy Hunt's & Dave Thomas's book *The Pragmatic Programmer: From Journeyman to Master*, I use a few editors and know them very well. You must know your editor well enough to do the same kinds of things to it that I have done to mine.

To obey *One Button Testing* and *Fault Navigation*, I want to run a batch file each time I hit one function key, and I want to navigate to faults and errors easily. One fast way for me to do this is write the batch file in C++, and run this program out of Visual Studio each time I change the code.

So I use C++ for my script language and Ruby for the production language. This is not because I'm setting a good example for the selection of my script language. It's only because I know Visual Studio better than any other editor, and I know how to use it to apply many of our *Fault Navigation* Principles.

We stop and start the `miniServer.rb` skin of MRW, each time we test. I want all this to happen under the <F5> button, because I use VC++ to edit Ruby. The simplest thing that could possibly work, unfortunately, is to write my build script not in Makefile language, but in the language that the current <F5> locally invokes.

This (*sigh*) is our "Makefile":

```
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include "stdlib.h"
```

```
#include "stdio.h"


// miniServer.rb <- select, then <Alt+Ctrl+G> to open

    int
main()
{

    system("killall ruby >>terminate.txt");
    system("start ruby -W2 miniWiki.rb localhost 8080");

    FILE *f = _popen("ruby miniWiki_.rb", "r");
    char zone [4096];
    bool passed (false);

    while (char *z = fgets(zone, sizeof zone, f))
        {
        fputs(z, stdout);
        if (strlen(z) > 8)
            {
            char *zap = strchr(z + 5, ':');
            if (zap)  *zap = '(';
            zap = strchr(z + 5, ':');
            if (zap)  memcpy(zap, ") :", 3);
            }
        if (strstr(z, "##########################################  "
                 "All tests passed!"))
            {
            passed = true;
            system("killall ruby >>terminate.txt");
            }
        OutputDebugString(z);
        }

    return 0;

}
```

I'm not revealing it just for the opportunity to implore you never to type it into a C++ environment. Because my least favorite editor runs that language when I hit the <F5> button, I can either waste time learning another editor, or I can use VC++ as a scripting language. The `_popen()` line calls the Ruby test rig.

The lines using "`zap`", midmost, turn `:99:` line numbers into `(99)` format. Then we buffer `_popen()`'s output stream into `OutputDebugString(z)`. VC++ navigates to such lines, on <F4> or <F8>, as errors or warnings.

(See the Case Study Model View Controller, on page 170, for more than you ever want to know about `OutputDebugString()`.)

Each time the script runs this…

```
system("killall ruby >>terminate.txt")
```

…the previous incarnation of our internet server vanishes. Deleting a web server out of memory may seem unorthodox. I consider it less so than writing a Makefile in C++.

The Remote Test Button

MiniRubyWiki's text file transclusion feature has a big button called "Test", because doing anything else from such a Web page is not a good idea. The Test button will invoke the command found between the bangs `!ruby -v!` in the Wiki source. We will supply a more useful command than printing out Ruby's version number.

(Our web pages could do anything we can specify on a command line, so don't put such a web page on the public internet!)

The requirement:

- A "**Test**" button can save the text file, then send its name to a console program.

The test:

```
def surfToPage(args)
    sampleFile = args.fetch('sampleFile', SampleFile)
    transcluder = "!text!"
    transcluder += args.fetch('command', 'ruby -v')
    transcluder += "!"
    transcluder += sampleFile
    writeFile('WikiTestPage.wiki', transcluder)
    surf("WikiTestPage")
    assert_equal(false, @ie.Offline)
    expect = /\<TITLE>Cannot find server\<\/TITLE>/
    assert_no_match(expect, htmlNode.innerHTML)
    ieWait()
end

def surfToSamplePage(args = {})
    sampleFile = args.fetch('sampleFile', SampleFile)
    writeFile(sampleFile, Hello_World)
    surfToPage(args)
end

def htmlNode()
    @ie.document.childNodes.length.times do |i|
      if @ie.document.childNodes(i).nodeName == 'HTML'
        return @ie.document.childNodes(i)
      end
    end
end

def test_testTranscludedText()

    file3 = "files/sample3.txt"
    zilchRemoteFile(file3)

# we need a very simple Test action. cp might not exist, copy is internal,
# and xcopy asks stupid questions

    cmd = "cp --verbose #{SampleFile} " + file3
    surfToSamplePage({'command'=>cmd})

    testButton = getNode(SampleFile, "Test")
    assert_not_nil(testButton)  #  TFUI
    assert_equal('submit', getAttr(testButton, 'type'))
    assert_equal('Test', getAttr(testButton, 'value'))
```

```
# @ie.Visible = true; exit 0

        testButton.click()
        ieRefresh()
        assert(remoteFileExists(file3), getRemoteFolder() + file3)
        assert(Hello_World, remoteRead(file3))

        ieWait()  #  TODO  ieRefresh() ?
        sleep(1.5)
        ieWait()

   # puts htmlNode().innerHTML
```

The output of the console program appears, formatted, below the `<textarea>`.

```
        assert_match(
            /files.sample.txt. -&gt; .files.sample3.txt/,
            htmlNode().innerHTML
            )
    end
```

Note that `surfToSamplePage()` grew the ability to accept arguments from its test case clients. Each needs different features in its sample pages. Our new test case calls it to produce a Wiki page like this:

```
 !text!cp --verbose files/sample.txt files/sample3.txt!files/sample.txt
```

I use the GNU `cp` command, not `copy` or `xcopy`, because `copy` is built-into CMD.EXE, and `xcopy` wasn't working.

A note about style: These tests still are not exemplary. They are too long, and they reach across too many modules to get things done. Short tests that call many fixtures would propel development, and would obfuscate a book's narratives.

On that theme, here's the code to generate the text file's `<form>`:

```
    def cgi_input(id, type, value, name = "")

        id += '?' + name  if id != ""

        return '<INPUT id="' + id +
                  '" type="' + type +
                 '" value="' + value +
                  '" name="' + name + '"/>' + "\n"

    end  #  because cgi.input is not <wellformed/>

    def transcludeTextArea(command, fileName)

        fileContents = ""
        fileContents = readFile(fileName)  if File.exists?(fileName)
        fileContents = CGI::escapeHTML(fileContents)
        command      = CGI::escapeHTML(command)
        fileName     = CGI::escapeHTML(fileName)
        result       = CGI::escapeHTML($result)

        contents = cgi.textarea(
                        'name' => 'contents' ) {
```

```
                    fileContents
                    }

    contents += '<br/>'
    contents += cgi_input("", "Hidden", $currentPageTitle, "returnTo")
    contents += cgi_input("", "Hidden", fileName, "fileName")
    contents += cgi_input("", "Hidden", command, "Command")

    if '' != command then
        contents += cgi_input(fileName, "Submit", "Test", "Test")
        contents += ' '
        contents += command
        contents += ' '
    end

    contents += cgi_input(fileName, "Submit", "Save"  , "Save")
    contents += ' '
    contents += cgi_input(fileName, "text"  , fileName, "NewFileName")
    contents += cgi_input(fileName, "Submit", "Clone" , "Clone")
    contents += '<br/><strong><pre>'
    contents += result
    contents += '</pre></strong>'

   return transclusionForm(fileName, getPrefix() + 'engaging', contents)

end
```

Notice I had to write `cgi_input()` to get an XHTML version of `<input/>`, with the `/>` on the end. The decision to use Ruby's CGI module to generator HTML is not working out, and future versions of the complete project shouldn't use them.

MRW's CGI input layers, like all CGI applications, convert HTTP GET and POST parameters into a map of names and values. Code in MRW's various CGI layers detects when the user selects the Test button, and calls this:

```
def commandTest(cgiParams)
    command = cgiParams['Command']
    f = cgiParams['fileName']
    c = cgiParams['contents']
    writeFile(f, c)
    puts("Engaging: " + command)  if ! $DEBUG
    result = ''
    result = `#{command}`  if command.length > 0
    return result
end
```

MRW's CGI layer trivially saves the file, runs the command, and paints the output into the next incarnation of the current page:

Similar code should support the Save button.

We have generated a prosaic CGI feature, with just a little XHTML, to prepare the way for the next feature. For only one step more complexity, it will yield incredible power and flexibility.

Extensible Markup Language

The web needed a text-based data format compatible with HTML but not bound to it. And the web provided the tools for its own distribution, so XML libraries spread easily. Then, even for non-web projects, XML filled a sweet spot—a light database format supporting both manual editing and arbitrarily complex relations.

An XML file is a hierarchy of one or more `<>` tags delimiting nodes. Nodes may have attributes, contents, and nested nodes. Parsers read the nodes to build a node tree in memory—the Composite Pattern. Support systems like XPath traverse this tree.

We added the ability to edit and process text files to MRW to get ready for the next features. (Planning ahead is perfectly Agile; if this were a real project, users could already be using our first primitive feature while we work on the next.)

The next features:

- The transcluded file can be **XML**.
- The XML shall **render** in HTML.
- Such HTML shall contain **edit fields** for each attribute and content area

We need Wiki pages that transclude XML when they contain a line like this:

```
!xml!ruby -v!files/sample.xml
```

The command we will eventually feed the XML into, `ruby -v`, is just a place-holder.

The test:

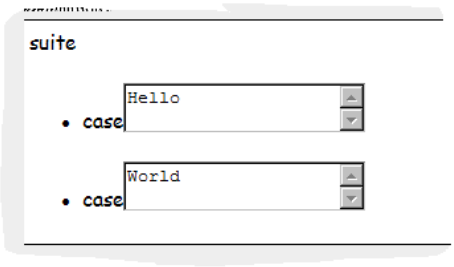
```
SampleXML = "files/sample.xml"

Hello_World_XML =
  '<suite>
     <case>Hello</case>
     <case>World</case>
   </suite>'

def test_transcludeXML()
    writeFile(SampleXML, Hello_World_Xml)
    transcluder = "!xml!!" + SampleXML
    xhtml = formatWiki(transcluder)

    assert_no_match(/#{transcluder}/, xhtml)

    doc = Document.new(xhtml)
    textareas = []
    XPath.each(doc, '//textarea') { |t| textareas << t }
    assert_equal('Hello', textareas[0].text)
    assert_equal('World', textareas[1].text)
end
```

A summary: `writeFile()` writes a sample XML file. `formatWiki()` converts the new line of Wiki notation into well-formed XHTML.

`assert_no_match()` ensures our code did not pass `!xml!ruby -v!files/sample.xml` through as clear text.

`Document.new()` creates a parsed XML node tree, and `XPath.first()` extracts its `<body>` element. `XPath.each()` finds each `<textarea>`, and the last assertions prove the XML payload appeared on the Web page as editable contents.

That test will force our code to generate Wiki content looking like this:



But what's the simplest way to convert XML into XHTML?

Extensible Stylesheet Language for Transformations

XSLT provides a flexible, powerful language for transforming XML documents into something else, such as HTML documents. But XSLT pushes the commonly understood definition of a "programming language". Template languages have both programming logic and string manipulators. Some, like Perl, put the logic on the outside and the string operations on the inside. Others, such as ASP or XSLT, nest logical operators inside string operators. This twist will challenge our refactoring skills, so tests are, again, mission-critical.

This new source file, `transcludeXML.xslt`, contains minimal XSLT to convert a trivial sample of XML into XHTML. Our Wiki will pass this XSLT and our input XML into a transforming module in an XSLT library. XSLT uses XPath, such as `select="name()"`, to read the input file and supply responses in its output.

XSLT with `media-type="text/html"` treats nested XHTML tags as output. This snip builds a `<form>` with the input XML node contents in `<textarea>` controls. To reduce your linguistic burden, I put grey on the `<html>` tags. Here's the minimum XSLT to produce the above XHTML fragment:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:param name="fileName"/>

  <xsl:output method="xml" media-type="text/html"
      standalone="no" omit-xml-declaration="yes" />

  <xsl:template match="*">
    <span>  <!--so the output fragment is well-formed-->

      <xsl:value-of select="name()"/>
      <ul>
        <xsl:for-each select="*">
          <li>
```

```
                <form name="{$fileName}!/suite/case[1]">
                  <xsl:value-of select="name()"/>
                  <textarea name="contents">
                  <xsl:value-of select="text()"/>
                  </textarea>
                </form>
              </li>
            </xsl:for-each>
          </ul>

        </span>
      </xsl:template>

    </xsl:stylesheet>
```

The output looks like this, formatted for clarity:

```
<span>
  suite
  <ul>
    <li>
      <form name="files/sample.xml!/suite/case[1]">
        case
        <textarea name="contents">Hello</textarea>
      </form>
    </li>
    <li>
      <form name="files/sample.xml!/suite/case[1]">
        case
        <textarea name="contents">World</textarea>
      </form>
    </li>
  </ul>
</span>
```

Ominously, this primitive XSLT hard-codes the names of the tags: "`/suite/case[1]`". That won't work when we adapt it to read any XML input. To watch this XSLT grow, we must first understanding its simpler versions.

The heart of the script's logic is "`<xsl:for-each select="*">`". It iterates over `*`, which is any node.

Tags without `xsl:` are XHTML outputs. So the `<li><lu>` tags create an outline, and a `<form>` tag brackets each `<textarea>`.

XSLT supports two ways to copy source XML into the target XML—one for outside other tags, and the other for inside. Outside a tag, `<xsl:value-of select="`*XPath*`"/>` evaluates the XPath (relative to the template's current node context) and inserts its result into the output. We use it for the `name()` of the outer node and inner nodes, and for the `text()` contents of a node.

But inside a tag, the first `<` delimiter makes the second `<` delimiter a syntax error, so XSLT provides a shortcut. `{`*XPath*`}` evaluates the XPath, and inserts its result inside an output tag. That's how we can set the `<form name="">` attribute to a concatenation of the `$fileName` parameter that MRW's functions passed in, and of the XPath to the current node.

Our Wiki must merge our input XML with that XSLT to generate that output. Your platform may have an XSLT transformation function that is easier than MSXML's, or harder, or both. For now, understand this complex function does the same thing as the command line `msxsl files/sample.xml transcludeXML.xslt fileName=files/sample.xml`:

```
def Transform(fileName, xsl, param = [])
```

```
        xslt = WIN32OLE.new("Msxml2.XSLTemplate")
        xslDoc = WIN32OLE.new("Msxml2.FreeThreadedDOMDocument")
        xslDoc.async = "False"
        xslDoc.load(xsl)
        return xslDoc.parseError.reason  if xslDoc.parseError.errorCode != 0
        xslt.stylesheet = xslDoc
        xmlDoc = WIN32OLE.new("Msxml2.DOMDocument")
        xmlDoc.async = "False"
        xmlDoc.load(fileName)
        return xmlDoc.parseError.reason  if xmlDoc.parseError.errorCode != 0
        xslProc = xslt.createProcessor()
        xslProc.input = xmlDoc

        param.each { |p,v|
            xslProc.addParameter(p, v)
            }
        xslProc.transform()
        return xslProc.output

    end
```

As usual, a Microsoft library supplies a great many powerful features that we must work hard to turn off.

Our `Transform()` method need only simple arguments. So here's the remaining code, in MRW's `formatWiki()` layer, to bond those files:

```
    def transcludeXML(command, fileName)  #  TODO  take a stream
        return Transform(
                    fileName,
                    'transcludeXML.xslt',
                    'fileName' => fileName
                    )
    end
```

Now we can walk through our new code, in order, and see what's going on. This code in a Wiki page…

```
    !xml!ruby -v!files/sample.xml
```

…triggers this bang detector within `formatWiki()`:

```
                if line =~ /^!xml!(.*)!(.*)/ then
                    return transcludeXML($1, $2)
                end
```

…which sends "`ruby -v`" and "`files/sample.xml`" into our XSLT, returns XHTML, and inserts it into the Wiki's output stream.

Now what if "`ruby -v`" were not a place holder but an important program? We are creating a Wiki that allows users to edit input, run it thru command lines, and display the output. So how can we respond to a button click on the client and run the target `command` on the server?

Shell to a Command Line

We pass the name of the transcluded file into the XSLT as a parameter because we need it in each `<form>` name, as:

```
<form name="files/sample.xml!/suite/case[1]">
```

The unique form name is the file name, a friendly bang `!`, and the XPath to our node. An XML with many nodes renders as many `<form>` tags, so each form's tag must refer uniquely back to its original XML node.The current design constrains our users not to put bangs or other oddities in their file names.

A sketch of our plan:

- pass `command` into the XSLT and store it in an `<input type="hidden"/>` tag in each `<form>`
- store the `$fileName` in a `hidden` tag
- store the unique XPath address of each `<form>` in a `hidden` tag
- add an `<input type="submit" value="Test"/>` button
- clicking it will HTTP POST the `<form>` contents to a new page called `engaging`
- that page will open `fileName`, use the XPath to fetch a node, write all the new attribute values into that node, and save them
- the page then invokes command, and adds the `fileName` and XPath to the command line
- the invoked command reads `fileName`, at the node XPath points to
- the command runs a test based on the attributes in that node
- the command writes test result data back into the XML file
- the Wiki commands your web browser to refresh the source Wiki page
- and you see the test results.

Most of those features are within reach, using common CGI techniques, and testage seen so far. Curiously, XSLT will provide a small speed-bump, around the middle of that list. It's good at using XPath, and annoyingly poor at generating it!

XSLT to Generate an XPath to any Node

We will soon add Clone and Test buttons to each `<form>`. When they invoke, they will need to know the correct XPath to their invoked test resources.

The code currently contains a familiar "temporary lie", to let the tests pass. We need a path to any node, not just nodes named `suite` and `case`, and each node's index should be correct. This XSLT is not good enough:

```
<form name="{$fileName}!/suite/case[1]">
```

It hard-codes too much. The `$fileName` comes from the Ruby side of MRW, but the XSLT side knows the names of nodes, and their indices. Let's write a test to improve the XPath component of the form names:

```
    def test_transcludeXML_xPath()
        writeFile(SampleXML, Hello_World_Xml)
        transcluder = "!xml2!!" + SampleXML
        xhtml = formatWikiTest(transcluder)
        doc = Document.new(xhtml)
form1 = XPath.first(doc, '//form[@name="files/sample.xml!/suite[1]/case[1]"]')
form2 = XPath.first(doc, '//form[@name="files/sample.xml!/suite[1]/case[2]"]')
        assert_not_nil(form1)
        assert_not_nil(form2)
    end
```

That leads to this new XSLT:

```
<xsl:variable name="xPath">
  <xsl:for-each select="ancestor-or-self::*">
    <xsl:variable name="idx"
      select="count(preceding-sibling::*) + 1" />
    <xsl:value-of
      select="concat('/',name(),'[',$idx,']')" />
  </xsl:for-each>
</xsl:variable>

<form name="{$fileName}!{$xPath}">
```

XSLT's expressiveness and clarity can rival Job Control Language (JCL). That code walks the "axis" of nodes from the current one to the root, accumulating the `name()` of each node. While MRW only supports two levels, that code is ready to produce deep XPaths, like "`/suite[1]/case[3]/row[19]/column[2]`".

The variable `xPath` also `counts` each node's siblings, to supply relatively correct indices like `[2]`. That system won't work when sibling nodes have different names; some indices will be too high.

MRW adapts to a useful range of potential XML. This XPath generates in only one place, and travels around to other modules as a string. No matter what arbitrary XPath we transport, test fixtures only use it to select their target nodes. They don't depend on how the query works. We can easily replace `xPath`'s value with different systems, such as matching a unique `@title` attribute.

When the Clone and Test buttons work, they will send HTTP GET commands containing that XPath. The CGI code that clones, and the test fixtures that test, will find their target nodes using that XPath.

Edit Transcluded Data

Now that our XSLT is more honest, we'll show some tests that added these and other features to it. The requirement is:

- Such HTML shall contain **edit fields** for each attribute or content area

The last section skipped ahead and did the content areas. Now we do the attributes.

Here's sample XML with an attribute tag:

```
<nodes>
    <node>Hello</node>
    <node attribute="value">World</node>
</nodes>
```

Following the spirit of XML, we put node contents into large `<textarea>` controls, and attributes values into small `<input type="edit"/>` controls.

This test drives Internet Explorer:

```
def test_transcludeXml()

    sampleXml = 'files/transclude.xml'
```

```
        xml = '<nodes><node>Hello</node>
                <node attribute="value">World</node></nodes>'

        writeFile(sampleXml, xml)

        transcluder = "!xml!ruby -v!" + sampleXml
        safeWriteFile(wiki('WikiTestPage'), transcluder)

surfToWebPage("http://#{TestSite}:#{$testPort}/WikiTestPage")

  #       @ie.Visible = true

  #   these lines force the XSLT to name the form after the XPath

        textArea = getNode( sampleXml + '!/nodes[1]/node[1]',
                            "contents" )

        assert_not_nil(textArea)

  #   this forces the title of the input field for an attribute to be the attribute name

        secondFormName = sampleXml + '!/nodes[1]/node[2]'
        secondForm = @ie.document.forms(secondFormName)

        assert_match( / attribute\s*:\s* \s*;/,
                      secondForm.innerHTML )

  #   now force the attribute target to be a populated text field

        editField = getNode(secondFormName, "attribute")
        assert_not_nil(editField)
        assert_equal('text', getAttr(editField, 'type'))
        assert_equal('value', getAttr(editField, 'value'))

    end
```

That produces the screen capture seen earlier, but the XSLT has become more dense to support it. We'll just reveal the code that supports attributes, and save the rest for later.

The test line `getNode(secondFormName, "attribute")` requires our XSLT to output an `<input>` field, like this:

```
<xsl:for-each select="@*">
  <xsl:text> </xsl:text>
  <xsl:value-of select="name()"/>
  <xsl:text>:</xsl:text>
 <xsl:text disable-output-escaping="yes">&amp;nbsp;</xsl:text>
  <input type="text" name="{name()}" size="31" value="{.}"/>
</xsl:for-each>
```

That says, "For each attribute in the current node, output a breaking space, the name of the attribute, a colon, a non-breaking space (` `), and an `<input>` tag named after the attribute (`name="{name()}"`) & containing its value in a value attribute (`value="{.}"`)."

For those few of you who remain insufficiently confused, yes we really are using the XML dialect XSLT to turn XML nodes containing attributes into the XML dialect XHTML

containing `<form>` nodes with `<input>` fields that represent the XML attributes as DOM attributes.

That has subtle benefits over writing all our XSLT features in a language like Ruby using brute-force procedural code. Because XSLT is XHTML-aware, users can write `&`, `&quot;`, `<`, and `>` marks directly into these `<input>` fields, and the XSLT will correctly escape them. The command `disable-output-escaping="yes"` suspends this feature when we output the non-breaking space code ` `. The `&amp;` in the string `&amp;nbsp;` is for XSLT's benefit, not XHTML's.

When I ran the test, it failed until I upgraded the XSLT. Then MRW gained the ability to display XML attributes as `<input type=edit>` fields.

Comments are Good

Some people who question Agile practices think we don't write source code comments. To the contrary—we put our comments on web sites. Frequent review reduces the odds they mislead some people.

Our next User Story is:

- Such HTML shall contain each XML **comment** in a *small, emphasized* font.

Users can't click on a comment, so we need only an XHTML Test:

```
def test_transcludeXml_withComment()
    xml = '<nodes>
            <!-- comment one -->
            <node><!-- comment two -->Hello</node>
            <node attribute="value">World</node></nodes>'

    writeFile(SampleXml, xml)
    transcluder = "!xml!ruby -v!" + SampleXml
    content = formatWiki(transcluder)

    doc = Document.new(cgi.body{content})
    e = XPath.first(doc, '/BODY/em/small')

    assert_equal(' comment one ', e.text())
end
```

`formatWiki()` is a middle-level function that takes Wiki notation and returns fragmentary XHTML. Inside MRW, the code calling `formatWiki()` will supply its frame and top-level XHTML tags.

The code `cgi.body{content}` ensures our XHTML fragment has a correct top-level tag, `<body></body>`. Otherwise `Document.new()` would report incomplete XML.

The XPath query "`/BODY/em/small`" returns the first small *em*phasized text it can find. The test asserts this contains the "`comment one`" payload.

That test forces one more change in our XSLT. Here it is in full:

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:param name="fileName" />
```

```
  <xsl:output method="xml" media-type="text/html"
      standalone="no" omit-xml-declaration="yes"/>

  <xsl:template match="*">
    <h3>
      <xsl:value-of select="name()" />
    </h3>
    <xsl:for-each select="comment()">
      <em>
        <small>
          <xsl:value-of select="." />
        </small>
      </em>
    </xsl:for-each>
    <ul>
      <xsl:for-each select="*">
        <li>
          <xsl:variable name="myPath">
            <xsl:for-each select="ancestor-or-self::*">
              <xsl:variable name="idx"
                select="count(preceding-sibling::*) + 1" />
              <xsl:value-of
                select="concat('/',name(),'[',$idx,']')" />
            </xsl:for-each>
          </xsl:variable>

          <form name="{$fileName}!{$myPath}">
            <xsl:value-of select="name()" />
            <xsl:text>---</xsl:text>
            <xsl:for-each select="comment()">
              <em>
                <small>
                  <xsl:value-of select="." />
                </small>
              </em>
            </xsl:for-each>
            <!-- put in attributes as edit fields -->
<xsl:for-each select="@*">
              <xsl:text> </xsl:text>
              <xsl:value-of select="name()" />
              <xsl:text>:</xsl:text>
              <xsl:text disable-output-escaping="yes">
&amp;nbsp;
</xsl:text>
              <input type="text" name="{name()}" size="31"
                     value="{.}" />
            </xsl:for-each>
            <br />
            <TEXTAREA name="contents" style="width:100%" ROWS="4"
                      COLS="80">
              <xsl:value-of select="text()" />
            </TEXTAREA>
          </form>
        </li>
      </xsl:for-each>
    </ul>
  </xsl:template>
</xsl:stylesheet>
```

The entry point is `<xsl:template match="*">`. We have only one entry point so far; the top of the XML file. In the test's sample XML, the top is `<nodes>`. We output that name and its comment, then `<xsl:for-each select="*">` traverses each child node.

At each child node, we output its name and comment, code that duplicates the behavior of the topmost node.

Some languages make refactoring easier than others. To perform a simple Extract Method Refactor in XSLT, we must start by learning what a method is.

It is a `template`:

```
<xsl:template match="comment()" name="comment">
  <em>
    <small>
      <xsl:value-of select="." />
    </small>
  </em>
</xsl:template>
```

That paints a comment into our `<em><small>` tags. The calling code finds each comment, and calls the `template` on it.

```
<xsl:for-each select="comment()">
  <xsl:call-template name="comment" />
</xsl:for-each>
```

One last refactor: Move the `<xsl:for-each>` into the template:

```
<xsl:template name="comment">
  <xsl:for-each select="comment()">
    <em>
      <small>
        <xsl:value-of select="." />
      </small>
    </em>
  </xsl:for-each>
</xsl:template>
```

After those refactors pass their tests, check the output:

The small *italics* make the comments look like commentary, not prompts.

Modify and Clone XML

Our next effort adds two buttons, and gets the first one working. The User Stories are:

- Each node has a "**Clone**" button, which clones the node.
- Each node has a "**Test**" button, which sends the node's XPath and file name to a command.

That means each `<textarea>` in the above output will have little buttons after it called "Test" and "Clone". Test authors clone individual nodes, and can upgrade passing tests into tests demanding new features.

MiniRubyWiki now allows users to view XML files, clone entries in them, and submit these entries in batches.

This generates a sample XML file. `getRemoteFolder()` permits test data in the local Web server's configured folder. `remoteWrite()` uses it's these statements could be shorter:

```
def generateSampleXML()
    sampleXml = 'files/transclude.xml'

    xml = '<nodes>
            <node>Hello</node>
            <node attribute="value">World</node></nodes>'

    writeFile(getRemoteFolder() + sampleXml, xml)
    transcluder = "!xml!ruby -v!"
    transcluder += sampleXml
    remoteWrite(wiki('WikiTestPage'), transcluder)

    return sampleXml
end

def assert_input(form, item, value)
    hidden = form.namedItem(item)
    assert_equal( value, getAttr(hidden, 'value') )
end
```

The `assert_input()` makes the case that creates those buttons and the Clone behavior slightly shorter:

```
def test_modifyAndCloneXml()

    sampleXml = generateSampleXML()
    surf('WikiTestPage')

# reach out to the first node's form

    xpath = '/nodes[1]/node[1]'
    formPath = sampleXml + '!' + xpath
    firstForm = @ie.document.forms(formPath)

# get the Clone button

    cloneButton = firstForm.namedItem('CloneXML')
    assert_equal( 'Clone', getAttr(cloneButton, 'value') )
```

```
        assert_input(firstForm, 'TestXML', 'Test')
        assert_input(firstForm, 'CloneXML', 'Clone')

    # Need hidden fields storing our file name, XPath location, and command

        assert_input(firstForm, 'name', sampleXml)
        assert_input(firstForm, 'xpath', xpath)
        assert_input(firstForm, 'returnTo', 'WikiTestPage')
        assert_input(firstForm, 'Command', 'ruby -v')

    # write new text into the <textarea>

        contents = firstForm.namedItem('contents')
        nuSample = 'new text'
        contents.setAttribute('value', nuSample)

    # clone the new text

        cloneButton.click()
        ieRefresh()

        xml = remoteRead(sampleXml)
        doc = Document.new(xml)

    # cloning node[1] puts its new sample text into node[2]

        e = XPath.first(doc, '/nodes/node[2]/text()')
        assert_equal(nuSample, e.value)

    end
```

That checks for hidden fields. They transmit enough data back to the server to clone or test.

To change the XLST to emit the new buttons, the method that calls xsltproc needs more arguments:

```
    def callXsltProc(fileName, command = '')
        return callXsl(
                    'miniWiki.xslt',
                    fileName,
                    { 'fileName' => fileName,
                      'Command' => command,
                      'returnTo' => $currentPageTitle }
                    )
    end
```

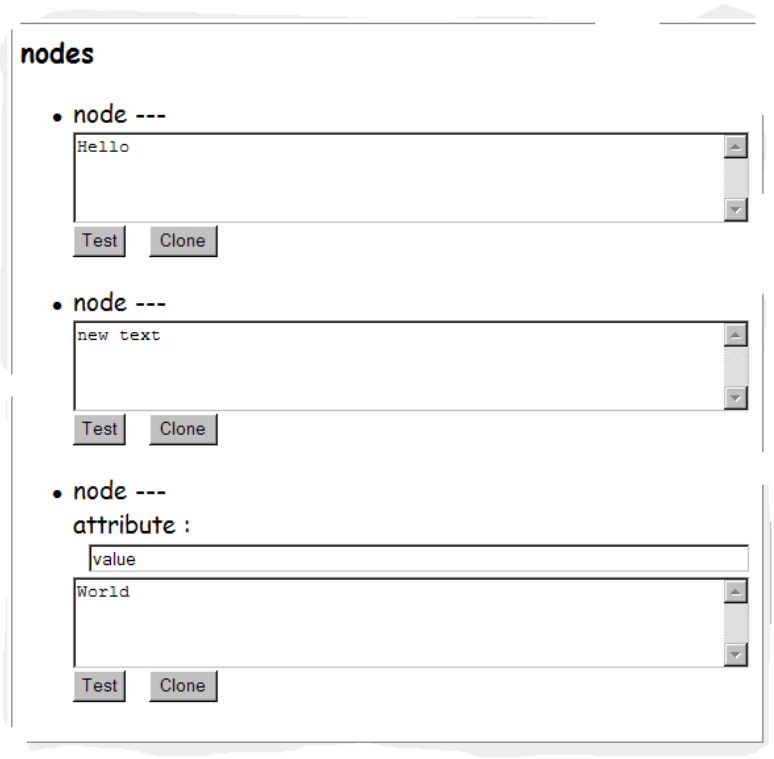
That map of parameters routes into `xsltproc`. We must call it on a shelled command line, so the parameters all get extra `"'quote marks'"` around them:

```
    def callXsl(xslFile, xmlFile, param = {})
        params = ''

        param.each { |p,v|
            params += "--param #{p} \"'#{v}'\" "
            }

        return `xsltproc #{params} #{xslFile} #{xmlFile}`
```

```
end
```

(The version of `callXsl()` in the online source checks if the competition, `msxsl`, is available, and might call it if `xsltproc` is not installed.)

We target this effect. The user writes new text into a node, hits Clone, and the node duplicates:



To get there, the XSLT collects its new parameters, and adds them to `<input>` nodes after each `<textarea>`:

```
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:param name="fileName" />
  <xsl:param name="Command" />
  <xsl:param name="returnTo" />
…
        <form name="{$fileName}!{$myPath}">
…
          <TEXTAREA name="contents" style="width:100%" ROWS="4"
                    COLS="80">
            <xsl:value-of select="text()" />
          </TEXTAREA>

          <input type="submit" value="Test" name="TestXML" />

          <xsl:text disable-output-escaping="yes">
&amp;nbsp;
</xsl:text>

          <input type="submit" value="Clone"       name="CloneXML" />
```

```
            <input type="hidden" value="{$fileName}" name="name" />
            <input type="hidden" value="{$myPath}"   name="xpath" />
            <input type="hidden" value="{$Command}"  name="Command" />
            <input type="hidden" value="{$returnTo}" name="returnTo" />
          </form>

...
</xsl:stylesheet>
```

They create hidden fields inside the <form>, so at HTTP GET time those data come into the Wiki through CGI. MRW's response code maps them in `cgiParams`, and this (somewhat procedural) code clones the node:

```
    if cgiParams['CloneXML'] == 'Clone' then

        fileName = cgiParams['name']
        xml = readFile(fileName)
        doc = Document.new(xml)
        node = XPath.first(doc, cgiParams['xpath'])

        nu = node.deep_clone()
        nu.text = cgiParams['contents']

  #   push in any modified attributes

        extraAttributes =
["contents", "CloneXML", "name", "xpath", "Command", "returnTo"]

        attributeKeys = cgiParams.keys - extraAttributes

        attributeKeys.each { |key|
            node.attributes[key] = cgiParams[key]
            }

        node.next_sibling = nu

        open(fileName, 'w') do |f|
            doc.write( f, 2)
        end

        $returnTo = cgiParams['returnTo']

    elsif …
```

`$returnTo` then uses the miracle of global variables to trigger code, elsewhere, that refreshes your Web browser back to your page, which now displays the cloned node.

Our transcluded XML pages clone nodes to facilitate test experiments. One should test by cloning a passing test, changing it a little, and seeing if it passes.

Test Transcluded XML

To decouple tests from the development process…

(Our last project, The Web, creates an environment to store a database of these resources. See page 404 for the last few steps preparing a Web browser to host the data, then sending them through their test fixture.)

Our XSLT converts arbitrary XML files, containing test resources, into XHTML with `<form>` tags, and `<input>` fields for each XML attribute. Users can edit test resources from any Web browser. Transclusion means "transitive inclusion", where data go both ways.

The next step sends that data into a test fixture, tests a hypothetical software project, collects the results, and displays them.

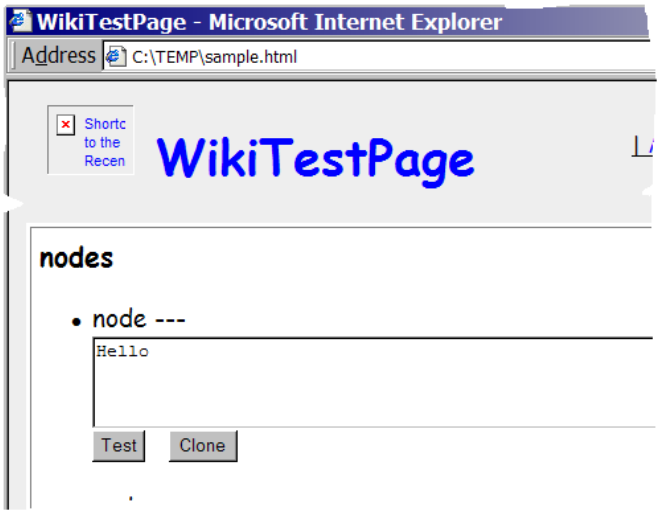
To make the next feature more challenging, its tests showcase a very nice HTTP Test framework called WebUnit, by Yuichi Takahashi. Your project (per the Minimize System Diversity practice) should use only the fewest test frameworks possible. This Case Study illustrates many of them. WebUnit, like HttpUnit, would make test-first as easy as our other frameworks, if our tests developed fixtures to support it.

Temporary Visual WebUnit Inspection

Our first test, naturally, is a *Temporary Visual Inspection*. `WebUnit::TestCase` inherits `Test::Unit::TestCase`, and `setup()` gets WebUnit ready to hit Web pages:

```
require('webunit/webunit')
require('win32ole')

class WebUnitTests < WebUnit::TestCase

    def setup()
        super()
    end

    def test_testTrancludedXml()
        sampleXml = generateSampleXML()
        url = "http://localhost/wiki.rb?WikiTestPage"
        response = Response::get(url)
        reveal(response)
    end

    def reveal(response)
        tempFile = 'c:/temp/sample.html'
        writeFile(tempFile, response.body)
        ie = WIN32OLE.new('InternetExplorer.Application')
        ie.navigate(tempFile)
        ie.Visible = true
    end
end
```

Our new version of `reveal()` writes a temporary file, surfs IE to it, and produces this:



Note the upper left corner contains a broken graphic instead of MRW's mascot, the PenBird. If we miss him, our tests should ensure the temporary test folder, `C:\TEMP`, contains `penBird.gif`. His `<img>` tag correctly defines his image size, so the broken graphics won't disturb our page's geometries.

MRW uses no other included dependencies, such as "Cascading Style Sheets". If your Web site uses them, your project must put their reference copies into your equivalent of the `C:\TEMP` folder for these kinds of Temporary Visual Inspections to work.

This version of `reveal()` does not comply with the *Temporary Interactive Test* Principle, because `C:\` is not a Web server.

Test Server Fixtures

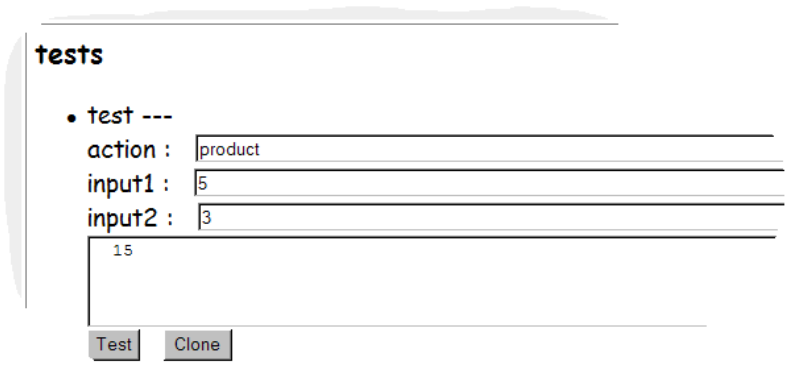
This test clicks the test button. Our target test fixture, until now, has been `ruby -v`. That would return Ruby's version information. We need a test fixture, in a command line, with these characteristics:

- Takes the XML file name on the command line
- Takes an XPath to the target node on the command line
- Returns the word Pass or Fail, so our XSLT can paint the node pink if it fails
- Writes a new `result` attribute into the XML.

This XML contains trivial sample tests. Two pass and one fails:

```
<tests>
    <test action="product" input1=5 input2=3>  15</test>
    <test action="product" input1=8 input2=2>  16</test>
    <test action="product" input1=4 input2=6>2004</test>
</tests>
```

Our project's strategy targets any XML, so long as it fits a few wide requirements. For example, if nodes only contain something short, like that little number "`15`", they should not use such a big `<textarea>`:



MiniRubyWiki's Test Transclusion system resembles Ward Cunningham's Framework for Integrated Testing (FIT), with significant differences. FIT has a Wiki interface, FitNesse, maintained by Object Mentors. It would display the above text as HTML output, not `<form>` fields to edit. Put another way, to change FitNesse's numbers, a test author hits EditPage, and edits data inside the Wiki page source. They appear in a simple markup language. A row of data looks like…

```
| 200 | 300 | 500 | -100 | 60000 | 0 |
```

…and FITnesse displays that in neat columns, like this:

eg.ArithmeticFixture					
x	y	x + y	x - y	x * y	x / y
200	300	500	-100	60000	0
400	20	420	380	8000	20

MRW, by contrast, puts the most frequent Customer Team actions (edit, test, and clone a test) directly onto the surface of each Web page. Programmers must assist writing test fixtures, so they must naturally assist generating new XML formats, placing these in a server, etc. Future versions of MiniRubyWiki, available at `http://rubyforge.org/projects/minirubywiki`, will continue to compete with FitNesse (at `http://fitnesse.org`) on these usability issues.

Read more about Acceptance Test Frameworks in the book *FIT for Software Development*, by Ward Cunningham & Rick Mugridge.

MiniRubyWiki targets any of a range of XML schema. This requirement causes some near-term issues, such as a huge `<textarea>` to store a single `15`. FitNesse, by contrast, uses a neat column. Future MRW examples will use the node as a case-specific comment.

MRW cannot use XML alone to determine whether to leave the big `<textarea>` out. XML cannot distinguish empty node contents from no contents, so future blandishments will read some metadata or other and coax these small data fields into readable tables. This Case Study targets big data outputs.

Our next activity is hitting that Test button (as usual). When it runs, it will invoke a fixture called "`fixture.rb`". Our test will write that (to avoid checking-in too many files).

`fixture.rb` collects its command line arguments, opens an XML file, converts it into a `Document` object, queries it using the `xPath` argument, pulls out the payload variables, and

```
require "rexml/document"
include REXML
```

```
xmlFile = ARGV[0]
xPath   = ARGV[1]

open(xmlFile) { |f|
    xml    = f.read()
    doc    = Document.new(xml)
    node   = XPath.first(doc, xPath)
    action = node.attributes["action"]
    input1 = node.attributes["input1"].to_i()
    input2 = node.attributes["input2"].to_i()
    expect = node.text.to_i()

    if action == "product" then
        if input1 * input2 == expect then
            puts "pass"
        else
            puts "fail"
        end
    else
        puts "fail"
    end
    }
```

Note the complete lack of error checking. Just a little more code could `rescue` this fixture from incorrect XML input. Low-level error checking would clutter a book about high-level error checking.

More code (not shown) lets `generateSampleXML()` output those sample files, and puts "`ruby fixture.rb`" into `WikiTestPage`:

```
!xml!ruby fixture.rb!files/transclude.xml
```

This WebUnit test hits the first test button, refreshes the page, and checks that the first node has a new field containing "pass":

```
    def test_testTrancludedXml()

        sampleXml = generateSampleXML()
        url = "http://localhost/wiki.rb?WikiTestPage"
        response = Response::get(url)

    #   spot-check the first form has a returnTo field

        form = response.forms[0]
        returnTo = form.parameters.find{|p| p.name == 'returnTo'}
        assert_equal("WikiTestPage", returnTo.value)

    #   click the Test button

        response = response.forms[0].submit( 'Test' )

    #   check the new response might be a refresh page

        assert_match(/refresh/, response.body)
        assert_match(/URL=wiki\.rb\?WikiTestPage/, response.body)

    #   refresh the page, allowing the server to finish the test
```

```
        response = Response::get(url)

    #   check the form's test passed

        form = response.forms[0]
        returnTo = form.parameters.find{|p| p.name == 'result'}
        assert_equal("pass\n", returnTo.value)

      #  reveal(response)

    end
```

In the CGI layer, this code passes the test:

```
    elsif cgiParams['TestXML'] == 'Test' then

        fileName = cgiParams['name']
        xml = readFile(fileName)
        doc = Document.new(xml)
        xpath = cgiParams['xpath']
        node = XPath.first(doc, xpath)
        node.text = cgiParams['contents']

        extraAttributes =
  ["contents", "TestXML", "name", "xpath", "Command", "returnTo"]

        attributeKeys = cgiParams.keys - extraAttributes

        attributeKeys.each { |key|
            node.attributes[key] = cgiParams[key]
            }

        open(fileName, 'w') do |f|
            doc.write( f, 2)
        end

        $returnTo = cgiParams['returnTo']
        command = cgiParams['Command']

        result = `#{command} #{fileName} #{xpath}`

        node.attributes['result'] = result

        open(fileName, 'w') do |f|
            doc.write( f, 2)
        end

    elsif …
```

That code duplicates the effects around Clone; it should naturally refactor together.

These statements, irresponsibly added to the end of that long test case, trivially check that the next two test pass and fail, respectively:

```
form = response.forms[1]
response.forms[1].submit( 'Test' )
response = Response::get(url)

form = response.forms[1]
returnTo = form.parameters.find{|p| p.name == 'result'}
assert_equal("pass\n", returnTo.value)

form = response.forms[2]
response.forms[2].submit( 'Test' )
response = Response::get(url)

form = response.forms[2]
returnTo = form.parameters.find{|p| p.name == 'result'}
assert_equal("fail\n", returnTo.value)
```

Similar tests can easily ensure failing tests turn their `<form>` pink. Low-priority features shouldn't slow down our narration. We can see a (green) light at the end of the tunnel. This Case Study is almost started.

WebInject

Acceptance Test Frameworks make test resources easier to write than test cases. We must check if MiniRubyWiki's framework is flexible enough to boost a Customer Team's productivity—long before the framework is finished. Putting someone else's XML and test fixture into it can verify its nascent abilities.

WebInject is an XML-based Acceptance Test Framework, written in Perl by Corey Goldberg, that targets HTTP servers. We will soon have an HTTP Test, using WebUnit in Ruby, that drives MiniRubyWiki to transclude XML written in WebInject's format:

```
<testcases>
  <case
    description2=
      'verify string &apos;Corey Goldberg&apos; exists in response'
```

```
    url='http://www.webinject.org/dev.html'
    output=''
    id='1'
    method='get'
    verifypositive='Corey Goldberg'
    description1='SAMPLE TEST CASE - load WebInject dev page'>
  </case>
  ...
</testcases>
```

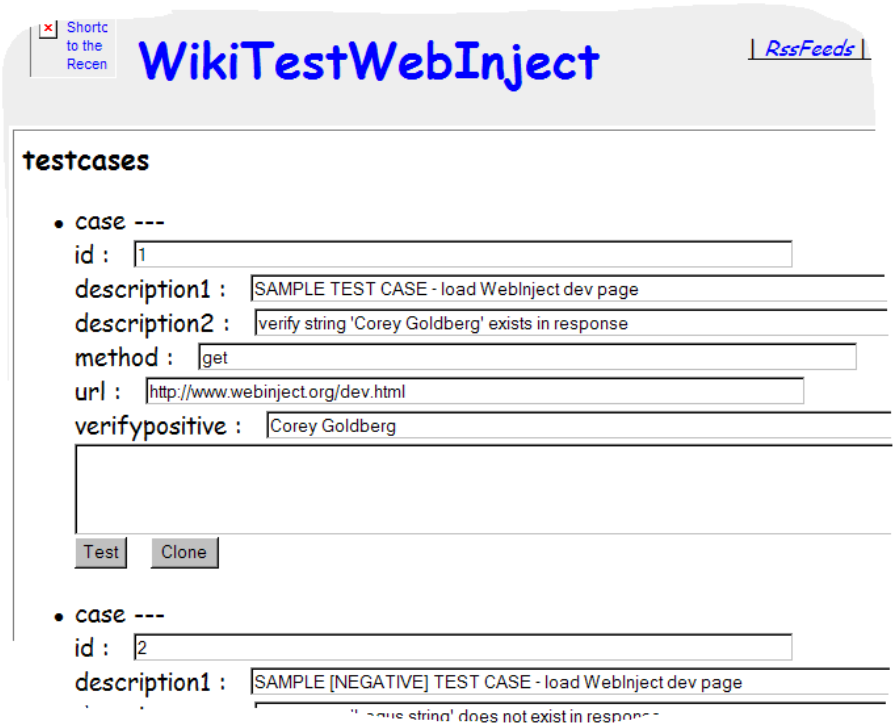
Our test case will click MRW's test button, and command MRW's server to shell to Perl, so this can evaluate `webinject.pl`. That script will operate an HTTP Test against Corey's Web server, collect results, and return them to our Web server, where our test will collect the final results, and our page will display them.

If those links sound fragile, that's the point. If you write a similar "Rube Goldberg" contraption, and fear it might break someday, then configure your test server to run it frequently and ring an alarm if it fails.

Configuring WebInject to trigger the MRW page that launches WebInject's MRW test is left as an exercise for the reader.

This test displays a Wiki page transcluding WebInject's sample XML, `testcases.xml`, found in the `./files` folder:

```
def test_WebInject()
    writeFile(
        wiki('WikiTestWebInject'),
   '!xml!c:/perl/bin/perl webinject.pl!files/testcases.xml'
        )

    url = "http://localhost:8888/WikiTestWebInject"
    response = Response::get(url)
    reveal(response)
end
```



As previously noted, XML with no use for its node contents cannot get rid of the large `<textarea>` yet.

Now our test hits the test button:

```
def test_WebInject()
    writeFile(
        wiki('WikiTestWebInject'),
   '!xml!c:/perl/bin/perl webinject.pl!files/testcases.xml'
        )

    url = "http://localhost:8888/WikiTestWebInject"
    response = Response::get(url)
    response.forms[0].submit( 'Test' )
    response = Response::get(url)
    reveal(response)
end
```

At submit time, MRW's CGI layer displays a temporary page saying "`processing...`" and containing a tag commanding your Web browser to pull the next page:

```
<META http-equiv="refresh" content="1;URL=WikiTestWebInject">
```

The second Response::get(url) call simulates a Web browser responding to that command and pulling the page.

MRW bonds `c:/perl/bin/perl webinject.pl`

With with the current XML file name and path, producing this:

```
c:/perl/bin/perl webinject.pl files/testcases.xml /testcases[1]/case[1]
```

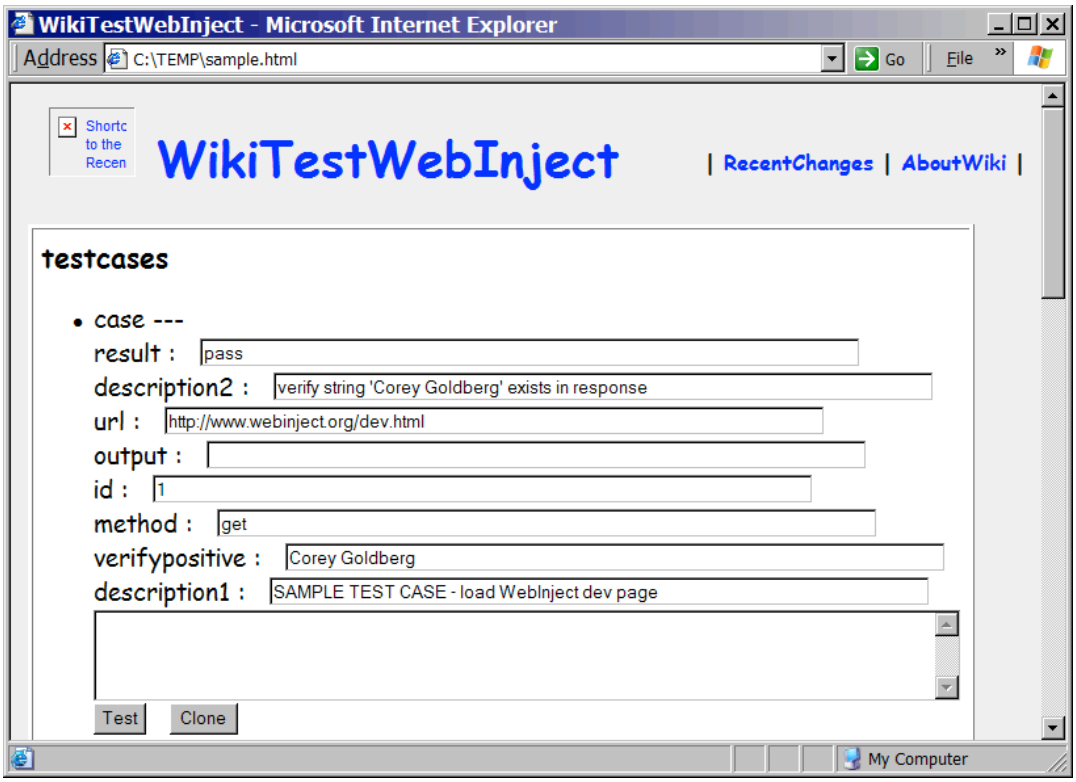
I added the complete path to ActiveState's Perl distribution (from `http://www.active-state.com/`) to avoid conflicts with other Perl distributions on my `PATH`.

`webinject.pl` hits Corey's Web server and verifies he wrote his name on it. (This is important—some Web site authors forget to do that!)

TODO This book cannot digress into all Perl issues. These functions, inside `webinject.pl`, that pull a Web page demonstrate the LWP library in action:

```
sub httpget {  # send HTTP request and read response
    $request = new HTTP::Request('GET',"$url");
    $cookie_jar->add_cookie_header($request);
    $starttimer = time();
    $response = $useragent->simple_request($request);
    $endtimer = time();
# elapsed time rounded to thousandths
    $latency = (int(1000 * ($endtimer - $starttimer)) / 1000);
    $cookie_jar->extract_cookies($response);
}
```

This version of `webinject.pl` outputs "`pass`" when the test at `/testcases[1]/case[1]` succeeds. MRW collects the output and puts it into a new attribute called "`result`":



These test fixture layers decouple the test resources from the test runner. If you execute "`perl webinject.pl files/testcases.xml /testcases[1]/case[1]`" from a batch file or command line, they test the same things, but they don't need any of MRW to work as their test runner.

Our next feature collects test output that Web browsers are designed to display.

Transclude Graphic Test Results

The Broadband Feedback Case Study ends (on page 281) with tests that copy GIF images into our `./files` folder.

People who might be too busy to install and run our application's bench version, every day, must chronically review these images.

They need pictures of their change requests delivered from our bench version to their Web browser. With this system, they could request a high-priority esthetic change, wait for a reply from programmers, hit their Test button, see the change, and confirm it. Use Broadband Feedback to minimize the time between specifying a new feature and delivering it.

To close that loop, we need a Wiki page transcluding XML test resources, a fixture that runs the tests and converts their outputs to images, and a little more code in our XSLT layer to display the images.

Previous test cases wrote the XML to transclude. This Web page transcludes a permanent XML file, found in our `./files` folder:

```
!xml!Project.bat!files/testProject.xml
```

`Project.bat`, in the Web server's home folder, contains the complete path to wherever we placed `Project.exe`:

```
C:\blah\blah\blah\Project\Debug\Project.exe --test %*
```

`files/testProject.xml` contains:

```
<tests>
  <test
    locale="en" listIndex="0" first_name="Ignatz" last_name="Mouse">
    Tests the first customer name, in the English locale
  </test>
  <test
    locale="sa" listIndex="1" first_name="Krazy" last_name="Kat">
    Tests the second customer name, in the Sanskrit locale
  </test>
</tests>
```

Those elements refer to controls on `Project.exe`'s dialog, which we will see shortly. Their test runner will switch to the given locale, click on an item in the list box, and take a picture of the resulting window.

The last feature in the Broadband Feedback Case Study, "Test Modes", on page 276, defined the fixtures used in this test case. It writes this to the console:

```
pass!http:files/TestDialogTestSanskritBroadbandFeedback.gif
```

MRW is going to split that into two new XML attributes—`result` and `output`.

MRW uses two skins—the Apache `httpd` server, and a built-in `miniServer.rb`. The former typically runs as a service, so it will have a hard time launching a test that raises a window. MS Windows imposes many restrictions on its hidden service applications.

Still using WebUnit, this test hits `miniServer.rb`'s HTTP server on Port 8888:

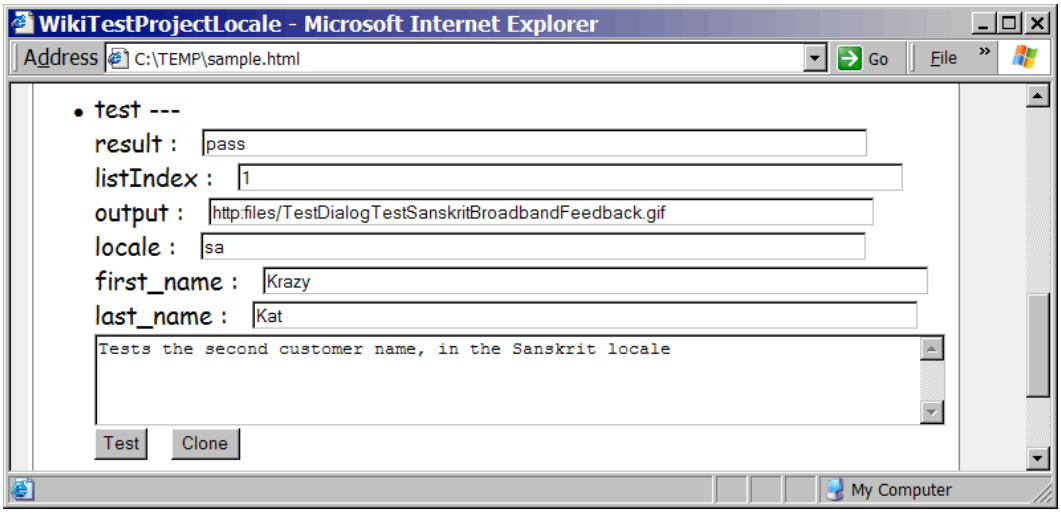
```
  def test_ProjectLocale()

      writeFile(
          getRemoteFolder() + 'WikiTestProjectLocale.wiki',
          '!xml!c:/phlip/Project/Debug/Project.exe '
               + '--test!files/testProject.xml'
          )

      url = "http://localhost:8888/WikiTestProjectLocale"
      response = Response::get(url)
      form = response.forms[1]

  #   hit the second test button

      response.forms[1].submit( 'TestXML' )

  #   refresh the page

      response = Response::get(url)

  #   find 'pass' in the result field

      form = response.forms[1]
      result = form.parameters.find{|p| p.name == 'result'}
      assert_equal('pass', result.value)

  #   find the address of a GIF file in the output field

      output = form.parameters.find{|p| p.name == 'output'}
      expect =
'http:files/TestDialogTestSanskritBroadbandFeedback.gif'
      assert_equal(expect, output.value)

  #      reveal(response)

  end
```

When a test fixture returns, this new Perl-style code splits its results and stores them into new XML fields:

```
result = `#{command} #{fileName} #{xpath}`
result =~ /(pass|fail)!?(.*)/

node.attributes['result'] = $1

if ! node.attributes.has_key?('output') then
    node.add_attribute('output', '')
end

node.attributes['output'] = $2  if $2

open(fileName, 'w') do |f|
    doc.write( f, 2)
end
```

MRW's CGI layer stores results within the XML resource. Test fixtures could write on it too, because nothing stores the XML in memory during all these transitions. Both the Wiki pages and fixtures read a fresh copy of the XML before using it.

Here's a snapshot of the result:



This system uses the node contents as a big `<textarea>` containing a comment. Different fixtures may find different uses for this area. A FIT-style test could put a comma-delimited data table there, and the fixture could parse it and run many tests.

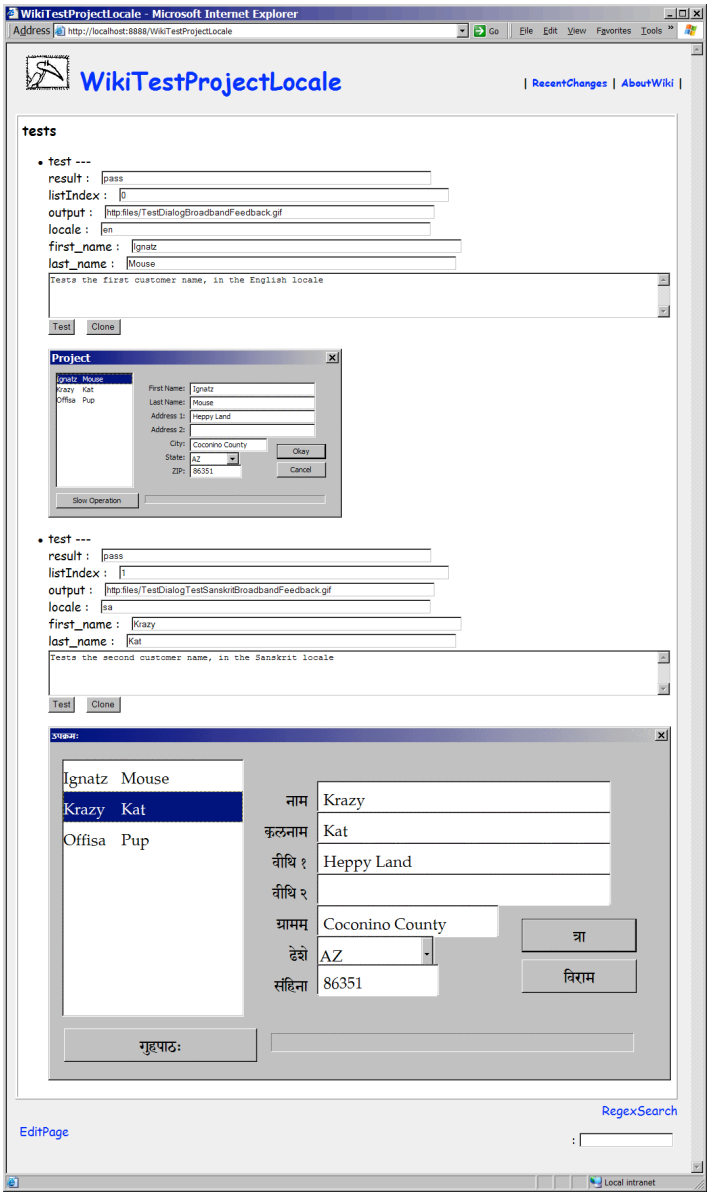
Our last flourish tweaks the XSLT. It must intercept attributes called "`output`", and transclude their contents as `<img>` tags. This Child Test, at the XSLT level, enforces this change:

```
def test_transcludeOutputImages()
    xml = '<tests>
            <test
                output="BrunoTheBandit.gif"
            not_output="don\'t transclude me"
                />
           </tests>'

    writeFile('scratch.xml', xml)
    xhtml = callXsltProc('scratch.xml', 'ruby -v')

    doc = Document.new('<body>' + xhtml + '</body>')

    not_output = XPath.first(doc,
                  '//form/img[@src = "don\'t transclude me"]')

    assert_nil(not_output)

    output = XPath.first(doc,
                  '//form/img[@src = "BrunoTheBandit.gif"]')

    assert_not_nil(output)
end
...
      <input type="hidden" value="{$returnTo}" name="returnTo" />
      <xsl:if test="@output">
```

```
            <p/><img src="{@output}"/>
          </xsl:if>
        </form>
```

...

Those three new lines of simple XSLT code turn our Wiki into a gallery of test output:



That Wiki page conducts Broadband Feedback, turning traditional GUI development inside out. Traditionally, programmers open a window, manually put data inside it, and see what that data does. This Wiki page puts the data on the outside, and automatically displays the window's animated results on the inside. We approach the Agile ideal of making tests extremely easy to write, and applying them to collect high-quality feedback from all aspects of a project. Not only its Logic Layer.

To Do

This Case Study finished its features of highest value—clone, edit, and test transcluded resources, and display their graphical outputs.

The living MiniRubyWiki project will support many more features, fitted to the topologies of the projects it assists. This list of suggestions applies to any Acceptance Test Framework:

- Error recovery, from each level:
  - broken XML in the XSLT translator
  - missing fixtures & command line errors
  - broken XML inside the fixture
  - test failures with a diagnostic message
- Clone an entire XML source file
- Edit XML raw, not rendered
- Test all resources in a file, all of a page, or many pages
- Color successful cases lime, failures pink
- A complete test hierarchy, with sub-nodes
- Expand or contract nodes
  - Node expansion state persists after test runs
  - Contracted nodes containing failing sub-nodes are pink
- Navigate to the tested case at refresh time
- A slideshow mode for each animated GIF, with VCR buttons
- Merge or bounce simultaneous attempts to change the same XML
- Cases can be marked as New, Passed, or Failed
- Automatic e-mail when a case moves from Passed to Failed. (This indicates an engineer must reproduce the failure in a code-level test.)

Those features are now easy to add.

Conclusion

Judge our Case Studies' successes by these criteria:

- We know how to start new features with tests
- To create new esthetics, we write tests sooner than use form painters
- To create new event handlers, we write tests sooner than use Class Wizards
- The more time spent tuning tests, the lower the risk of needing the debugger
- We can review usability without installing a new release.

This book has closed its loop. Further work in all these Case Studies consists of using existing test fixtures, and adding similar ones, to test new features into these projects.

Part III: Explications

Parts I and II described a strategy that assumes many software engineering concepts. This segment connects the strategy to the greater milieu of a healthy software lifecycle.

Chapter 13: *Agility*

The book *Lean Software Development*, by the Poppendiecks, explores the best metaphor for software Agility. “Lean Manufacturing” is Agility for hardware; the closest analog to software Agility. Software assembly costs less than hardware, as it’s weightless and proton-free. Source code’s ease of change provides many false paths away from good designs.

This essay mixes common metaphors: Driving, airplane takeoffs, hill climbing, etc. Like Peter Merel’s metaphor, “The Tao of Extreme Programming” (from the book *Extreme Programming Examined*), this chapter, alone, will not teach you how to implement and practice Agility. It shows what to expect and observe as a team follows specific techniques found in any of the dozens of fine books on these subjects. Metaphors are bridges between concepts; each leads to a different aspect of the same thing. The heartbeat of an Agile process is passing tests.

Feedback

Information, metrics, senses, knowledge, and results revealing a project’s current state, in real-time, are priceless. Software development is murky and mysterious by nature, and must force down the cost of high-quality feedback.

Development follows two nested cycles. Writing source code to implement features is the inner cycle, and delivering versions to users is the outer cycle. If feedback from tests drives the inner cycle, then feedback from users can steer the outer cycle more frequently and accurately.

The best feedback is profit from user productivity.

All software engineers endeavor to efficiently produce features that add value. Selling those features is secondary. Demonstrating the ability to reap the rewards of their value is primary.

Some projects don’t automate tests for all aspects of their code, so slow and expensive manual tests delay some feedback, and impede improving user productivity. Inner cycle delays compound outer cycle delays.

Some projects implement their features in order of technical risk, not business priority. That also delays improving user productivity. After the wait, users might request rework, adding to the risk of bugs. Outer cycle delays compound inner cycle delays.

When tests prevent long bug hunts, time estimates become short and useful, stabilizing software development’s inner cycle. Implementing features in order of business value stabilizes the outer cycle. Feedback prevents risk.

A project becomes Agile when the rate that feedback collects information about the current position exceeds the rate a team changes that position.

A car, driving on a dark stormy night, needs headlights that illuminate things farther away than the driver's reaction time and safe turning radius. Visibility relieves stress.

To lower the cost of change (and reduce that reaction time) an Agile project automates tests for everything that moves, and runs tests relentlessly. While changing a module, automatically test it after the fewest possible edits; say 10 at the most. Only perform small, incremental edits that immediately return a project to a useful state.

To lower the odds of reworking a project's features (and to boost those headlights), frequently release useful versions to users. But don't release bugs. Relentless testing trades long hours debugging for short minutes writing tests, permitting frequent, reliable releases.

Engineers leverage existing tests to reduce the cost of new tests. This book enables test-first for GUIs by introducing new behaviors into the cycle of test writing.

Oil and maintain your project's test machine, to lower the cost and boost the value of each test run, and to back up the team's courage. Testing is hard to learn. Any source code lacking tests delays feedback. Teams need experience, practice, communication, and ambition to automate a test for any kind of situation.

Communication

Projects written by more than one programmer can only scale when everyone communicates. The three general kinds of communication are writing, speech, and test cases. Each has costs and benefits, but some teams get their balance wrong.

Some teams carefully write design plans, hoping to prevent later rework. Such designs, once implemented, might resist unplanned changes. Projects that plan their designs before implementing them might reject ideas to improve designs that occur too late.

Some teams assign one module to each programmer. They plan narrow interfaces, take these plans to separate offices, write unplanned code for a long time, and then attempt **Integration**. Low-quality feedback is worse than none. Programmers who could not immediately take over each other's work cannot maintain a high velocity.

Some teams write excess paperwork to communicate. Some programmers pride isolation to defend their creative process (and their anxiety while debugging). So some projects plan their designs for a long time, and some projects reduce their paperwork burden by only planning interfaces, and allowing programmers to implement unplanned modules in isolation.

Test coverage makes inspiration, intuition, spontaneity, and collaboration safe. Relentless testing changes the kinds of communication a project can leverage to get things done. Everyone knows Agile teams reduce their paperwork burden, from requirements gathering to source comments. Common work areas, Shared Code Ownership, self-documenting tests, and laziness are only parts of the reason.

Time spent "perfecting" an object model before implementing it delays feedback, so use a "good enough" object model, and implementation procedures that frequently review the model and maintain its flexibility. Agile processes use constant team interactions to target a balance between too little and too much formality.

Agile teams set a very low threshold before changing code. Formal documentation could never hit a moving target. **Design Smells** are feelings or attitudes about code quality, which may or may not link to rational explanations. Engineers who smell them are required (in the presence of tests and reviewers) to refactor and follow their nose to any improvement.

When code experiences a series of changes, each leading directly to a passing test, no change can reduce the code's ability to pass tests. Some improve it. Refactoring helps entropy

favor robustness. New test cases become easier, and each makes the code's features less likely to break during a change.

Testing and refactoring work together to decouple objects from their neighbors, and cohere objects with their test cases. Test-First Programming is a design technique. Changing minimal code while passing tests uplifts that code's design quality. Some teams plan designs for testing. Agile teams design *by* testing.

Testing the act of changing code fundamentally shifts how our industry teaches good designs. Books like *Design Patterns* show good solution instances, frozen in time. Books about Agility assist the search for good designs, integrated with activities that pay for the search.

Tight, automated feedback, and high-bandwidth communication within a team, increase the odds of preventing errors. Unnoticed errors lead, later on, to long bug hunts in code that may have since grown more complex.

Agile teams complete features in small batches. Feature requests doled out a few at a time—tokenized as **User Stories**—prevent complexity. We only advance designs in small increments, fitting each new requirement in with the existing ones closely and irrefutably. Software engineers implement each User Story by writing its simplest abilities first, then coding advanced ones while refactoring their code together.

To begin a hard task, solve the simplest condition within it. To invent a language, make it `print("Hello World")` first. Solve for several simple cases, then merge the solutions to solve the general case.

When mathematicians write intricate proofs, such as for geometries that generalize to any number of dimensions, they first solve a specific proof for 2D, then a proof for 3D, then for 4D. Armed with these experiences, and proofs, they merge duplication into a single generic proof for any number of dimensions.

(Leaving only the multi-dimensional proof in the textbooks, to show off, turns math books into such light pleasant relaxing reading material.)

"Designing" means organizing the relations between the structure of objects in memory and their behavior in time. Structure is bone, behavior is muscle, and perfecting their design is difficult. The best way to seek simplicity is to start with it.

Simplicity

Designing reconciles conflicting requirements. Each conflict is an opportunity for complexity. The book *Notes on the Synthesis of Form* by Christopher Alexander presents the best rationale for designing in small increments. He compares self-consciously constructed mansions to humble dwellings, rebuilt yearly by their owners, with small adjustments respecting the climate and the lay of their land. Over-designing, to impress everyone, only satisfies the gullible.

In software, "small adjustments" means 1 to 10 edits before hitting the test button. Perform the smallest, simplest change you can that returns the project to a working condition, but with a slightly different design, or slightly more abilities. If someone improves code you wrote, rejoice unselfconsciously.

Designing on paper loosens the steering wheel (no tests to provide tactile feedback), darkens the headlights (no reviews of live code), and accelerates (when designs on paper are too easy to change). Planning designs generates very high rates of change with reduced

constraints, propelling our vehicle into the dark, toward a goal that might be wrong, and toward unseen obstacles.

Organic Design grows a solution, starting from the simplest case. A statement grows long, and becomes a block. The block sprouts into a function, which becomes a class. Crowded classes split by affinity, responsibility, and authority into modules.

This freedom can make adding features *too* easy. Feedback-driven systems need checks and balances to dampen wild perturbations, and inspire small adjustments.

As developers add features, the **Customer Team** roles review them, and censor any low-priority details that should not affect velocity. Agile teams often refer to the source of their marching orders as the "Onsite Customer", and the supporting roles (marketing, testing, domain, business, usability, etc.) as the Customer Team.

Customer Team leaders are authorized to request ambitious features when they accept the responsibility to monitor and analyze those features' growth into live code, and to suggest simplifications and corrections. In exchange, developers are authorized to grow source code as they see fit, if they accept the responsibility to make features easy for Customer Team leaders to specify and monitor. Projects grow **Customer Tests** to provide custom feedback for each User Story.

All experienced programmers remember diligently working on features, with complex internal logic and complex external displays, which users then never used. High-quality feedback and aggressive **Scope Control** reduce the odds of wasting time, both inside the application's internal logic and outside its GUI. Frequently demonstrate your new features to your Customers, as you create them, and when they tell you a feature is now finished, don't argue.

Only implement features the Customer Team has scheduled for the current iteration.

Customer Teams sort User Stories by business value.

A project must finish its primary set of features before working on a secondary set. For the first few iterations, only the primary ones get any design attention, coding attention, or tool support. Reviewing an iteration's results assists adding new User Stories to the stack, and assists re-sorting the stack before the next iteration.

Some projects start by guessing which tools they will need, then buying them. Even if an organization already bought a given tool, adding it to a project is still a purchase, expecting a cost of ownership lower than return on investment. If you delay an acquisition until simple code reveals the need for it, you might never need to acquire. If you do, your well-factored code will specify the exact tool you need, and will insulate other modules from the change. After replacing a module, all your tests must still pass.

Delay expensive decisions until the last cheap moment.

Some Agile literature admits a diagnosis of nebulous or rapidly changing requirements indicates an **Extreme Programming** prescription. This sophistry appeases those with positive experiences converting relatively motionless requirements into planned designs before implementing them. But we don't care if all the requirements are carved in granite.

Source code supporting the primary features, written first, experiences the most test runs while writing all subsequent versions. Finished primary features assist specifying new

secondary features, so their usability reinforces the primary features. New versions lead users toward the features of highest business value. All roads lead to Rome. Refactors invest secondary features into the primary features' code, amplifying the testing pressure that constrains the primary features.

Implementing features by business priority is a design technique.

Without tests, adding new features destabilizes old code. So without tests, we can't promise our Customers that they can safely request *any* feature next. Without tests, we might need to see *all* the requirements, to work on the hardest technical problems first. Without tests, the Customer can't easily steer in real-time toward a business goal.

Courage

Without test-first and refactoring, clients think they must assemble as many program requirements as they can afford to have written. This effort snarls all relative business priorities together, making Scope Control impossible. It obscures opportunities for simplification. Designing and implementing many features all at once is very hard, leading to our industry's reputation for very large failures. Studies have shown that the volume of requirements gathered before a project starts indicates that project's failure risk.

Putting tests in front of development's inner cycle permits an outer cycle of incremental feature growth. That relieves the Customer of the responsibility to predict the future and guess which complete set of features will maximize productivity.

Traditional teams who write tests last, after writing the target code, duplicate effort. They design for testing, then debug to remove the bugs that tests would have prevented. They lose opportunities for tests to drive design. And they may duplicate effort by collecting metrics to see how many different paths the tests cover. When tests force every path to exist, coverage metrics stay high as a side effect of rapid development.

When tests drive development and make changes safe, the search for the set of features that maximizes users' productivity becomes a hill-climbing algorithm. The Customer fearlessly picks the steepest slope from the current location. This simplifies requirements gathering. While details for low-priority features remain unclear, the high-priority ones compel attention. Implementing those features teaches how to specify the lower value features, so they support the higher ones.

Relentless testing over incremental changes enable Agile projects to:

- accept feature requests in any order, at any time
- release any Integration to **Quality Control** and beyond
- minimize the time between deciding a feature's specifications and using it.

A project's client steers with feature requests, getting the most important ones first. The sooner these features help users add value to their own endeavors, the sooner our project lifts off the runway and sustains itself.

Teams must remain technically and mentally prepared to fearlessly change anything.

The Extreme Programming literature revolves around a list of 12 practices, forming a minimal but safe process. Some projects need more than those 12. Doing those minimal practices will teach if your team needs more practices sooner than doing more or different practices will teach that you should only do the 12.

Many fine books describe these practices in sufficient pragmatic detail to coach teams to use them. (Many books will soon cover each practice.) This book focuses on how the practices, or their equivalents, support Test-First Programming for GUIs.

Extreme Programming

The practices are:

- **Test-First Programming**—upgrade a feature only if you can fail one test case
- **Simple Design**—provide minimal production code to pass the current tests
- **Refactor Mercilessly**—change any code in any module for any reason at any time
- **Continuous Integration**—the whole team always uses the same source
- **Frequent Releases**—regularly finish a **Production Build**, and push it out of the lab
- **Customer Tests**—literate tests that specify when each User Story is finished
- **Common Code Ownership**—nobody can tell who wrote what code
- **System Metaphor**—jargon linking the domain to the code by a common theme
- **Pair Programming**—two engineers on one design task
- **Whole Team**—developers and Customer Team liaisons work in the same area
- **Sustainable Pace**—sleep, slack, and snacks are excellent design techniques
- **Planning Game**—a meeting to sort pending features by business priority.

Achieving those practices, together, requires behaviors that other methodologies explicitly declare, so planning, analyzing, architecting, reviewing, and documenting are all here, too. Each practice, performed alone, causes trouble. No practice is new. All the practices together reinforce each other's benefits.

I sorted the practices so the top six matter to individual programmers (whether their teams know they are eXtreme or not). This book's Principles enable those practices, which in turn enforce the promises of the remaining business practices. All those practices' complete descriptions are mercifully outside this book's scope.

GUIs interfere most with Test-First Programming, so the chapter of the same name, starting on page 443, completely describes its cycle, without GUI interference. Chapter 6: *The TFUI Principles*, on page **Error! Bookmark not defined.**, introduces extensions to the cycle to counteract GUI issues.

Use all the practices to prevent stress and risk. The top six practices reduce the cost of change, and the lower six reduce the odds of spending time recovering from an early mistake in requirements.

Time to Market

The only "evidence" that XP "works" is the rate programming shops learn to profit from Agility. Honest businesses are systems for collecting scientific data—they just call it "money".

XP addresses these common risks to velocity:

- long bug hunts
- delays before releasing

- reworking previously delivered features.

To speed development while preventing long bug hunts, XP recommends Test-First Programming, which treats the lack of an ability as a minor bug, and writes a test to capture that bug before killing it. The supporting practices are Merciless Refactoring, to paint yourself into corners and then cut new doors; Simple Design, to bypass excessive engineering efforts; Pair Programming, to learn from each change; Common Code Ownership, to minimize political or esthetic reasons not to change code; a System Metaphor, to shorten sentences, and Continuous Integration, to merge code changes before they conflict.

To avoid delays before releasing, XP teams do not just release often, they Release Frequently, typically every week, rain or shine. (Some releases don't deliver to real users. Each one need only demonstrate a hypothetical productivity boost.) Whole Teams know each release's status non-verbally, and have the mandate to automate much of their workflow. The business side reviews and extends Customer Tests to learn exactly what features appear in each release. Teams who Frequently Release must face disturbing issues, like installers or databases, and incrementally develop their scripts.

To avoid rework, XP teams boost user productivity early and often. The Planning Game sorts User Stories in order by business value. This is a hill-climbing algorithm—a search for the maximum productivity gain from the current position. On hills without secondary peaks, the shortest path up is always the steepest path from each point. In the space of programming, a hill-climbing algorithm encounters no secondary peaks when all application features deform continuously. Simple Design, Merciless Refactoring, and Test-First Programming change code smoothly and safely.

The fixes for those risks overlap. That's the point: The XP Practices are all "best practices" when used alone—in moderation. Put together, they bind and reinforce each other, permitting low overhead and extreme velocity. Questions about practices have answers in other practices. A Sustainable Pace also prevents long bug hunts.

Add Features not Modules

During a project's growth, the code always supports its current feature set with the fewest design elements and fewest source statements possible, no more. Groom the code to preen out **Cruft**; this frees the schedule, and lowers the defect rate. Compete to add features elegantly.

Developers gifted with the ability to plan a design that could cover all potential features are welcome here, if they hold that complete design up for a goal but not the path. Teams expect members to specialize, and to teach others how they do what they do. Nobody "owns" any module, and pairs switch frequently.

A feature is not finished without its GUI. At all times a program should form a useable round-trip application. And at all times a GUI should comfortably permit a user to drive all of a system's current features, without gaps.

Developers volunteer to complete User Stories that affect modules they understand. No module grows in isolation without immediately satisfying other modules. Each User Story requires changes to any number of modules.

Libraries are modules for sale. Application-specific modules are skeletons of libraries, presenting clean interfaces over only those few features their applications need.

Modules reluctantly grow more complex as a project matures. Complexity that is easy to add becomes hard to remove, so discipline constrains our creativity.

Emergent Behavior

Consider this house rule:

- Anyone putting things in the trashcan, which stick out over the top, must take out the trash.

As a metric, it balances behavior. Nobody struggles with too much in the can. Nobody appoints the job of taking out the trash. No fixed roles or schedules.

That kind of discipline produces instant feedback. If someone feels they take out the trash too often, or others less, only then needs the team add another rule (or a bigger can).

Apply distinctions to transitions between states to emerge complex behavior. A "distinction" is a definition that continuously and unambiguously defines membership. Any two people who know the distinction can independently declare the same set of elements to be its members, without checking each other's results. A "heuristic" is a distinction used to select a path. Heuristics, not phases, govern chaotic processes.

Anyone can tell whether something in the trashcan sticks out over the top, and what to do about it.

A process is "sensitive to initial conditions" if infinitesimal differences in input cause huge divergence in output. A butterfly flapping its wings in Africa might create (or inhibit) tropical depressions in the eastern Atlantic that travel west to become hurricanes. The design process we seek is the reverse—*in*sensitive to initial conditions. We want robust, flexible code regardless of major differences in input—such as the order in which our flighty Customer Team requests features.

The trashcan is always ready for the next user, whether the last user supplied many small things, or a few big things.

Engineers apply simple but distinct rules to transitions between software states to tune and focus results. Simple rules create complex engineer behaviors that, in turn, create simple, flexible, scalable, and maintainable code.

Metrics

Some software projects impersonate a hardware manufacturing practice called "Statistical Process Control". Developers pause every few minutes to measure and record data about their workflow, behaviors, and results. Every day they roll those data up into complex inscrutable metrics that detect rates and trends. Those indicators lead project decisions. In manufacturing, that technique works great to keep your tools sharp.

Software design is not hardware manufacturing.

Agile teams, in a common workspace, keep their most detailed and important metrics up in the air. Excessive formal data collection would slow Whole Teams down. Many metrics, such as the "Project Velocity", are consumed at the point of production. The Planning Game calculates the previous week's velocity, and immediately schedules an equivalent volume of stories for the next week. All these metrics have as much effect on a process as formal ones.

Agile processes produce copious tests, code, and releases, all with raw data ready for experts to efficiently collate into metrics. Few Agile shops bother.

Agile processes say, "If something is not working, you will get hard evidence indicating what, very soon. We expect the complete lifecycle to work at all times, so any perturbation gets noticed." Call that one big fat obvious omniscient metric—like an overflowing trashcan.

Why should statisticians claim the "works all the time" signal is not a metric? Because it leverages subjective impressions and feelings, beyond hard evidence. Disregarding intuition is illogical. Intuition assists our most important design goal, simplicity.

The Simplicity Principles

Simplicity is the opposite of complexity. It does not mean, "Turn your brain off," or survive with simple-minded code. That grows tangled and complex. Simple reductions enforce elegance. Battle-hardened engineers may have trouble getting simple, so XP promotes Pair Programming to compete over the most elegant solution. When an acolyte pairs with a guru, both learn.

Source code is the ultimate distinction: It either compiles and passes tests, or it fails. At the heart of any heuristic algorithm are simple rules to prune false starts.

> After adding each small ability, run this checklist, in order, and refactor until your code:
>
> 0. Obeys your team's Sane Subset and style guidelines
> 1. Passes All Tests
> 2. Expresses Intent Clearly
> 3. Duplicates No Behavior
> 4. Minimizes Classes, Methods, and Statements.

Repeat those checks, throughout a project's lifecycle, to keep it simple but understandable at all times. Apply them in order—don't remove that last bit of duplication if the result wouldn't express intent clearly. Some languages enable more obfuscation than others do. At least put duplicated things next to each other.

Simple code is easy to test. Tested code is easy to simplify. These extremes reinforce each other. Minimal Test cases force narrow dependencies between modules. Then tests permit refactoring, and squeezing code down to a minimum. If you try to remove some detail of complexity and make a mistake, the tests that required the features that required the complexity should stop you.

A car has brakes to enable going fast. The best way to go fast is to know exactly when and how to stop. Programmers use Test-First Programming to automate a system to trigger red lights. Writing tests prevents "process waste" when each one helps you go a little faster.

Before observing and introducing TFP, we observe, in painful detail, its opposite, the well-known and popular process that Steve McConnell's survey, *Rapid Development: Taming Wild Software Schedules*, calls "**Code and Fix**".

Chapter 14: *Sluggo at Work*

Sluggo has programmed since his early teens, and as his résumé reaches the 6-year mark, his youth lets him work the long hours that aggressive positions require of him. He carefully follows industry trends, such as ActiveX, Java, SQL, & XML, and his employers know he can tackle a wide range of technologies. Sluggo's **Muscle Memory** stores the right settings for a huge number of library functions.

His current employer supports dentist scheduling and billing with a series of Web pages. The clients requested a desktop version, so Sluggo must now write all the same user interfaces in Visual Basic. The backend, an SQL database, remains the same, as do the tier servers that provide business logic.

Sluggo knows his employers must approve his forms' layouts as early as possible, so he'll have time to process change requests. Early one morning, with the jungle rock turned up loud, he researches the Visual Basic online communities and locates a conversion script that reads a database SQL view and writes a VB form. It turns each column in the view into a label and edit field of the proper type and format. The form's source consists of a header full of specific metadata, and then a page of boilerplate code with some of the metadata's keys repeated into it. Most functions in that automatically generated code do not make any assumptions about the metadata, and many do not directly access the GUI.

Do's and Don't's

Sluggo then calls that script on each of dozens of database views.

> The worst design mistake here was not immediately fixing all duplication while that's so incredibly simple. The current code obeys (its own) style guidelines, passes all manual tests, and expresses intent clearly (because someone else wrote it). But Sluggo made it fail "Duplicates no behavior", and naturally "Minimal classes, methods, and lines".

The boilerplate code in each form is nearly all the same. Merged together, it could have become a complete and minimal Representation Layer. Replacing duplication with abstractions increases code's extensibility. Sluggo missed the opportunity to simplify every form simultaneously; at the very time any project needs the most extensibility.

A healthy Representation Layer would have emerged from the Simplicity Principles, even if only by accident.

A programmer unaware of Object Oriented Design concepts, but knowing a language very well, could have applied the *Duplicate No Behavior* Principle right now and finished in <25% of the time.

Instead, Sluggo spends time authoring the most important forms; editing labels and dragging controls around, to match their layout to the legacy HTML forms.

"Progress" Metrics

Sluggo congratulates himself that line count exceeds 2,000, before he ever executes any of them.

He connects a form to a sample database, runs the program, edits the form's data, submits them, and manually inspects the database. As expected, the data went into in the correct table, but another corresponding table did not change in lockstep. The quickie forms address the database SQL views, not the business objects in the middle layer. No problem. Sluggo already has "all" the forms painted; each one works well enough to support writing data that others will read, and each form has identical code ready for search and replace.

Sluggo has only "tested" once, manually, in the first several hours of production work.

He reads the server-side Java to see how it called the business object layer, and selects the "verify address" operation to convert first. He opens the customer address form and rewrites all the boilerplate code to match the Java code in the legacy application—roughly 30 lines changed. Then he sets a breakpoint in the new code, tries to run, fixes a few syntax errors, and steps the debugger into the new function.

Programming in the Debugger

Examining each line, he observes the code connects to the tier server properly; it creates a "`CustomerAddress`" object properly, and it passes an address in. Then he discovers the validation failed; the address was one big string with linefeeds in it, instead of four little strings. Instead of reading specifications, including the previous codebase, Sluggo uses bugs to teach requirements.

Without stopping the debugger, Sluggo writes new statements that pass the string through the `Split()` function. Then he uses "Set Next Statement" on a line above the new code, and runs it again. His editor lets him debug, edit, change variables, and manipulate the program's execution context all at the same time, while the program still runs. He discovers several more errors like this, and fixes them all without stopping the debugger.

Again, in several more hours' work, Sluggo has "tested" only once more; a big long manual test, while changing the code at the same time. Because he knows how to use all these risky and fragile debugging techniques, Sluggo considers himself a senior software engineer.

Duplicity

After this form works, Sluggo applies a little search-and-replace, and a little copy-and-paste. He changes all the generated code in all the forms, switching them from calling the database to calling the business objects.

Now not a single form works.

Over a couple weeks, wielding the debugger, he beats lumps flat on each important form.

Tools from reputable vendors contain heavily marketed and promoted features that support every one of Sluggo's techniques. His education and culture tell him to do things like this; nothing checks or balances this behavior.

Because he nearly always edits while debugging, and stops manually testing when the debugger clears the last line, he almost never runs the code through its complete round-trip in just one state. He always changes it along the way.

As he proceeds, he develops a database with predictable sample records, and he spools this into a backup file. After running a few manual tests, it will grow too dirty to continue; then he manually restores it. He leaves the database's administration window open in the background; each restore requires only a few keystrokes.

He also gets permission to search the Internet for demo versions of OCX controls. He selected a short list of ones that looked good, then found a reason to use each one. These controls came with bugs that destabilized the **Integrated Development Environment** (IDE). He put them all in at the same time, and remains unsure which ones crash, but he can't take the controls out now. Sluggo memorizes several sequences of events that lead to crashes, to avoid them.

Each time he begins a programming session, he sets a breakpoint and hits his IDE's Run button. Then he operates several forms to reach the breakpoint. His path avoids both known crashes and many simple known bugs that he plans to fix during the "cleanup phase". Getting to a target form feels like tiptoeing through a minefield.

So to avoid wasting that effort, he edits as much as possible before the editor forces a restart. He knows the catalog of edits that halt the debugger, and which activities might crash the IDE before he can stop a run and save the source. (Visual Basic 6 permitted changing code while debugging, but the editor disabled the Save button for no reason.)

Early in Sluggo's career, he discovered VB's `Debug.Assert` technique slowed him down. If it failed during a debugging session, it stopped the debugger on the assertion. But if he commented the assertion out, to make it stop failing, VB halted the debugging session. And it frequently and embarrassingly failed on colleagues' workstations. Sluggo avoids that statement.

Passing the Buck

A few weeks into the project, his boss requests to look at his forms. Sluggo schedules the appointment for tomorrow, to prepare. Then, he (begrudgingly) demonstrates the forms running on his own workstation, and his boss knows not to ask permission to drive the forms himself. Sluggo does not submit them to his office's version control system because "they are not ready yet". When he finally does, the forms crash the test computer, and he spends an evening and then a morning debugging there instead of at his workstation.

When the marketing department finally reviews the forms, they find a very long list of bugs and missing features. Nobody in Sluggo's company questions this high defect rate; they consider it a fact of GUI programming life, and they secretly multiply any time estimate that a "gooey programmer" gives them by three.

After fixing a bug, Sluggo searches for similar code in other forms that will contain the same bug. But with so many forms, he often misses the parallel bugs. When he finds and fixes them, he doesn't always manually test the parallel forms. So these "fixes" often introduce new bugs.

Manual Regression Testing

As the weeks get long, and as the functions get longer, every few days he manually checks each serious bug in the project's dead bug list, to confirm it remains dead. His department operates an elaborate bug tracking system (independent from the marketing department's feature request system), and it assists this behavior by printing out the dead bug list sorted by severity.

Sometimes a dead bug comes back to life and bites. Some bug fixes knock loose previous fixes, re-exposing the original bugs. Sluggo has taken to patching functions by testing their return values for bug conditions and then replacing the return value with the correct one, instead of going inside the function and killing the bug. New patches often contradict old ones.

Searching and replacing code to introduce fixes also causes regressions. When his search parameters accidentally differ from the duplicated target code, he misses some forms. Manually fixing them further diverges the statements. Over time, sloppy fixes for easy bugs make the hard bugs harder to isolate and fix. Each bug fix carries the risk of "domino bugs", where related modules incorrectly depended on the bug.

Sluggo also repeatedly tweaks and cross-patches the tables of metadata. The script that converted SQL views into forms listed every column, even the useless ones, and Sluggo naturally pruned many of those out. Then special cases required him to add some back in with more `If` statements.

Every time the marketing department receives a new version, it brings new bugs. They know they occupy the "quality assurance" role for the project. The marketing department representatives do not tiptoe through Sluggo's minefield. They manually test features the way users will need to use them until disgusted, and return a list with around the same number of bugs in it each time. Sluggo neglects proactive manual testing as he leans on these bug reports, and fixes each exact bug, oblivious to any possible similar bugs. The time spent debugging eats into the time he could have been adding features, or improving the internal structure to resist bugs.

Over a few weeks Sluggo experiences the trend he awaited—the rate at which new live bugs appear on his workbench finally ramps down. He knows this descent predicts the end date. But he knows not to announce the trend until it engages for a few days.

Just a Display Bug

The conversion script generated controls in alphabetical order. When Sluggo re-arranged the fields, he neglected their tab order. In fact, to edit each field during tests, he always grabbed the mouse and moved its pointer to each field to click into it and type. Sluggo never noticed the <Tab> problem, but manual testers, trained in the lowly task of actually using GUIs instead of the elite station of programming them, discovered the problem very early.

They used <Tab>, not the mouse, to move between fields. They noticed that the default field in each form was always the one whose database field name occurred lowest in alphabetical order, and that the <Tab> key took the input focus to fields whose database field name ascends in alphabetical order. To a user, the keyboard focus starts in the wrong edit field, then <Tab> jumps the focus all over the form.

> Most bugs would have been trivial to fix, not even requiring tests, had the source supported only a few leverage points; not hundreds.

Sluggo's boss gave this bug a very low priority, so Sluggo only gets around to it during his "cleanup phase". He arduously fixes each main form, one by one.

Before he can fix the minor forms, his boss declares a feature freeze. Sluggo expected this, and expected not to be consulted. Marketing's remaining feature requests must wait for the next version; Sluggo will only spend time removing high-priority bugs affecting data. The minor forms will release with their <Tab> bugs intact. Users who learn to use the main forms for data entry, without taking their hands off the keyboard, will each encounter an annoying surprise when they first use a minor form.

Again, the corporate culture supports these activities, and considers a "feature freeze" more than a month before the release a "best practice".

Experienced readers can reconstruct the last-week and last-night behaviors here.

Post Mortem

Of course we know a lack of automated tests hurt the most. Rampant duplication hurt a lot. A minor sample code repository or a major tools vendor could have supplied the script that generated Sluggo's initial source code. Either way, it brought an architecture that Sluggo should have questioned. Instead, he treated it as an officially ordained aspect of his environment, and worked within it. The effects of his first inappropriate change burdened the entire project.

A third style of development, different from both Code-n-Fix and Agile, specializes in carefully planning designs, then translating them into source. All these kinds of development have roles. However, planning designs can amplify the effects of early mistakes, made when programmers knew the least about a project. When Sluggo cloned and modified all those forms, he made a very big, very bad design decision without immediately verifying the decision with working source code. But even the best laid plans of mice and men oft times go awry.

After cloning his code, if Sluggo had removed duplication from all those forms, the design would then have fewer places to go bad. A few easy Extract Method Refactors would merge the boilerplate code into a single reused module. Even manual tests could ensure such simple changes worked. Fix problems early as a design grows.

Instead, Sluggo performed a "Create Bugs Refactor" on the innocent original code: Duplicate a function dozens of times, then change each cloned function, one by one, each change for a different reason, without testing many of them after each edit (remember all the search-and-replace?). Sluggo churned all the latent abstractions into obscured and tangled cruft. He turned a high line count into a high bug count.

Sluggo's working cycle chronically increased the difficulty of code changes. He tested manually, sporadically, and intermittently. He grew some modules in isolation from others, leaving obvious external features broken for long times. This prevented others from working on or reviewing the project.

Sluggo often displayed the GUI, but never in a reproducible state. He always ran the program into a situation manually first. This work loop depended on Sluggo's own ability to drive his broken GUI; no one else could have worked on the code.

Because his initial shotgun technique created more bugs than features, at no time did his system ever run bug-free. Sluggo rolled a snowball of bugs up a snow-covered hill, and each step made it bigger.

Chapter 15: *Nancy at Work*

Nancy began her career in technical writing. The available programmer role models hacked all night, ill from caffeine, then chased bugs for weeks. Chronic anxiety did not inspire her.

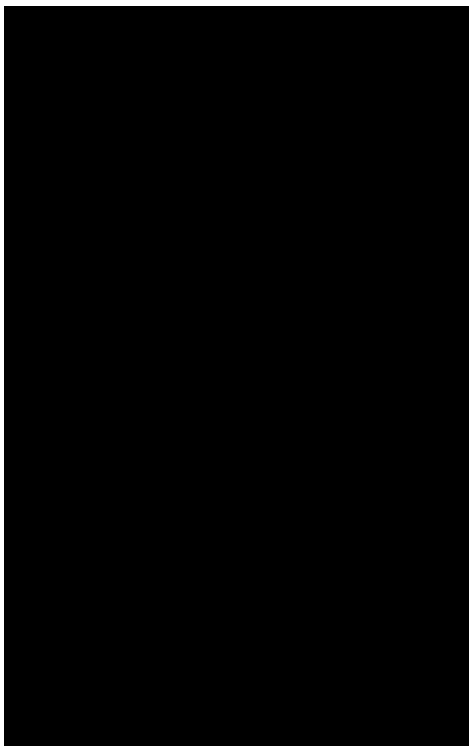
On writing assignments, she traded hoagies for technical details, and engineers programmed as she watched. On embedded firmware projects, with secret, invisible, and terse components, engineers designed for testing. Nancy's attendance transitioned into participation before anyone realized.

Her colleagues call her "the quiet one". She listens to a team's activities, and sees through industry fads and self-proclaimed paradigm shifts, providing a reputation for putting out fires before they start.

Her new company charts soil contamination. Factory owners sample underground chemicals at various locations, and store results in a warehouse of data. Sets of analysis results generate isopleth (contour) diagrams that reveal the chemical plumes' shapes and intensities at various depths. Environmental regulations require factories around the world to use software with these capabilities.

Extract Algorithm Refactor

An obsolete legacy FORTRAN program plots the isopleths. Nancy must convert both its logic and appearance into a modern canvas that illustrates internal mathematical activities as interactive reference lines. Users will drag those reference lines around to set the formula's coefficients, and see the isopleths instantly change. The new production code won't use a line of the old program.



Nancy's team has already selected Ruby (by Yukihiro Matsumoto) for the implementation language and the TCL/Tk Canvas as the drawing surface. This canvas treats each display element—lines, rectangles, text, etc.—as records to manipulate. The TkCanvas wrapper (by Hidetoshi Nagai) treats each record as a high-level Ruby object.

Each element can register callbacks from various keyboard and mouse actions. The canvas supports the concept of a "selected" element, and it can change any batch of element's attributes. Programmers use this to provide selection emphasis, such as a thick green dashed outline.

After accepting the assignment, Nancy wrote quick-n-dirty code to learn the objects and methods involved. Then she pushed this code out of her production folder,

and made an example of it. Eventually she'll recycle those files. Her experience making pancakes has taught her to throw the first one away.

The legacy FORTRAN program reads a table of XYZ coordinates, a bounding box, and a table of math constants; and it outputs a table of lines. The program's optional components rendered these into legacy formats, such as raw plotter commands. A professional mathematician wrote the FORTRAN source; some functions are 25 pages long, but the output is beautiful.

Three kinds of variables drive these charts—input data, cosmetic values and mathematical coefficients. The raw input is a table of XYZ locations, and a chemical intensity at each location. The outputs are isopleth diagrams of estimated intensities occupying flat planes through the ground.

Cosmetic variables affect display features, such as the distance between isopleths, their colors, weights, labels, etc. Math variables affect the relations between sample points and the estimated points between them. Some settings will blur a picture, while others will raise suspicion over subterranean anomalies.

In the present version, these pictures are read-only. Users must change configurations on another window, then return to this one to inspect the result. This forces users to keep program variable states and options in their heads.

Users have requested the ability to click and drag "grab handles" directly on these pictures, to change the input variables. Nancy's boss, both for licensing and performance reasons, suggests Nancy completely re-write the isopleth system, at the same time as she enables it for interactivity.

Early one morning, with the jungle rock turned up loud, Nancy manually runs the program in its simplest possible configuration. It takes a single data point and neutral constants, and outputs a single rectangle—the isopleth of a flat plane.

She then copies her manual experiment into a Ruby test case that sets the data up, shells to the FORTRAN program with `popen()`, collects the results, and asserts they are a single rectangle at the correct `Z` altitude. This new case only tests the legacy code.

She starts a log, on paper, of each task, and updates it at 30 to 90 minute intervals. As the project develops, she will forget this log; it is a technique to "prime the pump", and ensure the project starts correctly.

The first test took only ~30 minutes to write, and ~30 lines of code. The time between test runs will never be longer for the duration of the project. From now on, after editing 1 to 10 lines of code, Nancy hits a single button on her editor that saves all files and launches the Ruby interpreter. This runs a test rig, and passes the results back to the editor.

Her next test case creates a `TkCanvas`, draws a rectangle on it, and asserts that it now contains a rectangle. This is, of course, foundational for more aggressive abilities to come. Because the case never calls `Tk.mainloop()`, no window ever pops up, and Nancy never sees this rectangle. Tk is just another library, and `TkCanvas` is just another object.

Nancy's next test passes the same data point and neutral constants into a new Ruby function, `Isopleth()`. After the test fails, Nancy goes inside the new function and writes trivial math statements to calculate a rectangle that clips the bounding box at that `Z` altitude.

The test for this new function next calls the legacy FORTRAN program with the same data. The test then compares the old output with the new output. Because the assertions compare

floating-point numbers, she writes a function called "tolerance" to avoid **Hyperactive Assertion** failures when floating point numbers round off differently.

A hyperactive assertion fails when a user would think the downstream functionality was correct (but delivered on time). The first line of defense is tolerance:

```
abs(Subject – Reference) < (0.001 + abs(Subject)) / 100
```

Now she writes another test that calls the legacy FORTRAN program. It passes two XYZ locations, and receives a ramp. Nancy briefly tests that the return value consists of a smooth dip from one point to the other. (A "dip" is the 3D equivalent of a 2D "slope".)

After the tests pass, Nancy notices duplication in all the tests that call the legacy FORTRAN program. She merges the common elements into a **Test Fixture** that subsequent tests will then reuse. With the first primal test, her code's first small ability is almost ready to demonstrate.

Now the legacy FORTRAN program challenges the new `Isopleth()` function. To figure out how to upgrade the new function, she takes time to debug the FORTRAN program, alone, and see what it's doing with the math. Huge functions tend to conflate many special cases together, but Nancy eventually isolates the lines that process her simple input. She copies them out, places them in the Ruby code, translates their language, and makes the new test pass.

Then Nancy experiments with those statements, attempting to remove any detail that is not helping tests pass.

The Logic Layer can now produce a ramp, which the GUI Layer cannot yet do. Nancy adds a test; it demonstrates that two XYZ points passed into `Isopleth()` will appear on the canvas as a ramp.

Nancy adds a few more prototypical math operations. Along the way, to view her canvas temporarily, she often adds `Tk.mainloop()` at the bottom of a test case. Test runs then display a window, providing visual confirmation that no bugs are leaking through any blind-spots in her elaborate assertions.

Nancy finds herself frequently adding and removing `Tk.mainloop()` to tests. Each time she closes the temporary window, Tk crashes during the next test, because its internal systems do not typically expect to create any more windows after `Tk.mainloop()` exits. Nancy must frequently comment-out those `Tk.mainloop()` statements, then run all the tests again, just to make sure.

Nancy puts `Tk.mainloop()` inside a reusable function called `reveal()`, then researches how to prevent Tk from crashing when Tk exits `mainloop()` exits and then creates a new window. This information is rare, because most GUI applications start one event queue and leave it running until the application quits.

Nancy eventually locates `Tk.restart()`. This permits `reveal()` to contain enough features to safely temporarily reveal a tested window. The test fixture does not interfere with other windows under test.

Nancy's next feature leverages this fixture. She needs a test to simulate a user clicking on the bounding box. But she does not know exactly how to `bind()` a mouse click to a rectangle. She writes an experimental test, binds a disposable function to the mouse click event (`'Button-1'`) of the bounding box, and adds `Tk.mainloop()` to it. She runs the tests, a window appears containing a primitive isopleth, and Nancy clicks on it. The temporary code, in the test, outputs a trace, confirming how `bind()` works.

After exploring these options she'll add automated tests that preserve the effects she wants. She writes a function called `click()`, which simulates user inputs correctly on behalf of tests. Then she'll comment-out the `reveal()` call, so tests run silently.

Spike Solution

The program does yet have any complex features, such as multiple layered translucent colored isopleths synchronized with aerial photographs. Nancy wrote the minimum functions needed to "close the loop". Her test cases apply simple inputs and outputs to each point in the loop, matched the new code's behavior to the old code. The legacy program provides reference output, the new `Isopleth()` function provides matching sample output, the new Canvas displays this output, and users who click on the Canvas can change the input data.

In the jargon of program design, she analyzed the problem top-down, but depth-first instead of breadth-first. (The classic definition of top-down is breadth-first.)

> Nancy drove a "spike" down through all the program's potential layers, providing a minimal or acquired implementation for each one.

- GUI Toolkit bought
- GUI Layer renders isopleths
- Representation Layer manipulates isopleths
- Logic Layer generates isopleths

She spent a lot of time writing test fixtures in each of those layers, and she'll spend more time growing them, to ensure new tests are increasingly easy to write.

Nancy did not write (or generate with a wizard) hundreds of lines of code before closing this loop. For every remaining feature on her list, she visits each node in the loop, adding abilities. All new abilities get test cases. This code, still under 200 lines, is demo-ready, esthetic, and tested. Nearly all the planned features are missing, of course, but all the current behaviors are expected and predictable.

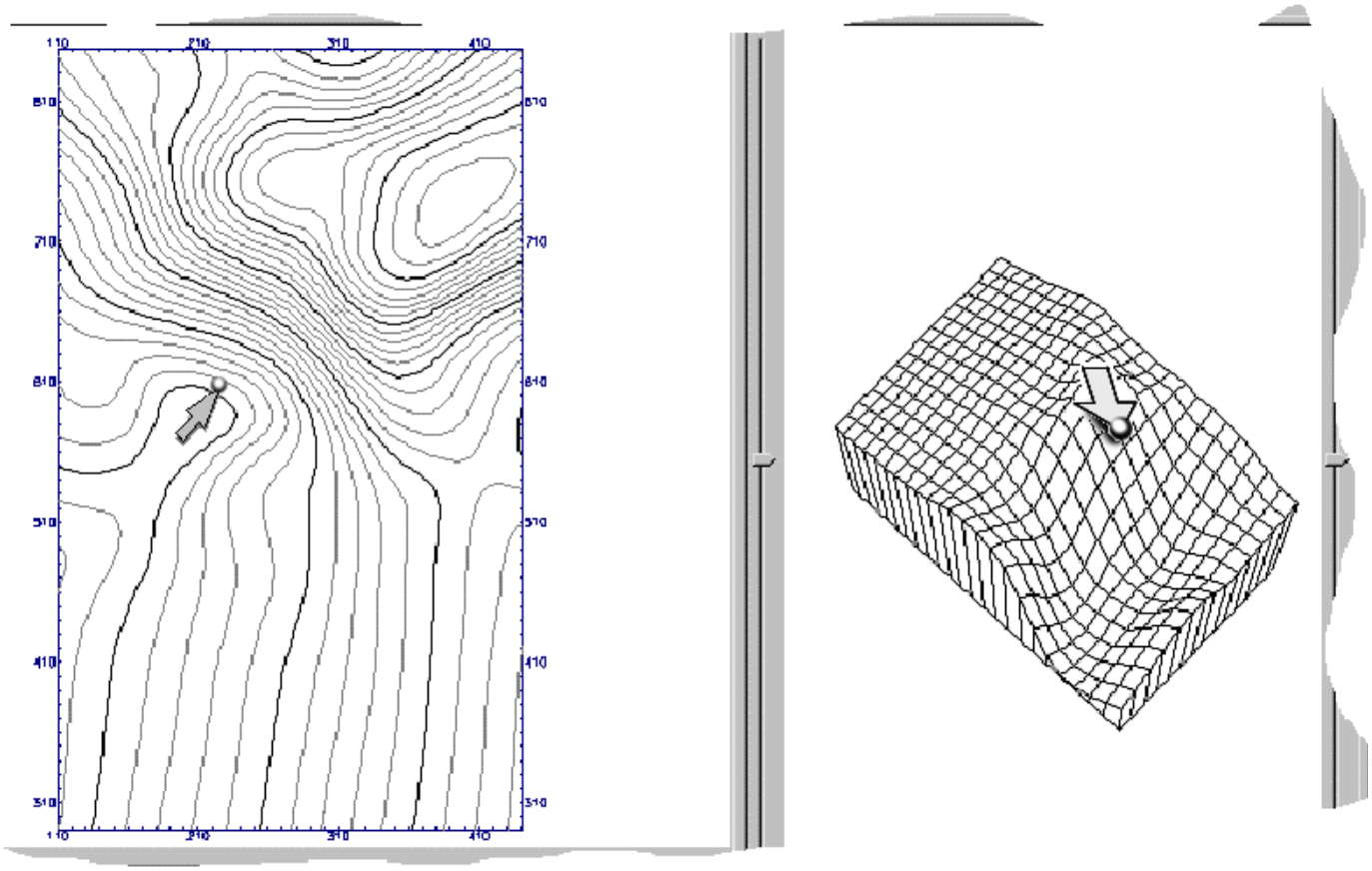
Before the end of the first day, Nancy integrates her new source into her team's version control system, runs it on another workstation to make sure it missed no dependencies, and demonstrates the primeval features to her boss, a marketing executive, and a quality control engineer.

> How many of you did not realize the last several pages were only the first day?

The crufty old Logic Layer needed a rewrite. It could not upgrade as rapidly as more modern code. Incremental tweaks would throw good money after bad. The re-write will channel the needs of the GUI Layer into direct real-time variable manipulations. Animations require an event-driven Logic Layer that—although it never imports any GUI Toolkit identifiers—supports algorithms carefully tuned to enable instant GUI responses.

Continuous Feedback

Over a few weeks, Nancy incrementally adds mathematical formulas to the Logic Layer, metrics and calibrations to the Representation Layer, and controls that display and change all these variables to the GUI Layer. The longer the project runs the better its tests constrain it, and the easier and safer changes become.



In the new interface, each data view integrates interactive controls. Users drag them to manipulate complex math coefficients. Here, one sets a bias angle in the interpolation of isopleth levels by dragging a ball mounted on the tip of an arrow. The arrow pivots around the highest point in the display, because the bias affects how slopes around this point appear. Both the isopleth & mesh views, and a numeric edit field (not shown), animate the bias angle's value changing in real-time, as the user drags.

Whenever Nancy completes an important feature, she demonstrates it to the quality control department. They research the math involved, and mine customers' databases for sample data, to write more tests. They enter the tests directly into a custom user interface, without delivering them to Nancy or other programmers.

Nancy's Acceptance Test Framework runs from a Web site. Users viewing that site can read tables of inputs and expected results. A column in each test resource presents the latest test's actual output. Web pages edit the input and expected outputs in data entry forms, each with a Clone and Test button.

Numerous tables cover every math formula and every display system, and many of their combinations. These tests run from top-level Web pages with names like TestAnisotropicKriging, TestIsopachKriging, TestKrigingWireframeMap, TestLinearDipProjectionTransectMap, TestTrendSurfaceResiduals, TestWeightedAverageSemiVariogram, etc. Cloning a table on one of those pages creates an identical test. Anyone on the intranet, even remote business partners, can clone a working test, change a variable, evaluate the new test on the server, and see instant results.

Many of these tests return images of user interfaces in action. The test evaluates behaviors in Nancy's canvas, and records screen captures of the results. Tests on interactive features—such as the relationship between an isopleth's bias angle and the little ball on the tip of that arrow—return an animation. Anyone in Nancy's company can write a new test to change that angle in concert with any other varying input data. The test server will return an animation of the ball and arrow rotating around the peak of their isopleths. In that animation, as its bias changes, the isopleths update in real-time. These Web pages have inspired many usability upgrades.

Each time Nancy hits her One Test Button, her test suite runs all the low-level tests that directly access her source, and many of the high level tests. If someone edits her site to add a test, and it passes, her next test run might include their new test case.

Nancy regularly reviews the code's design—often changing it with a colleague—to find ways to shrink code statements and improve structure. She makes sure that others can work on "her" module while she works on other, unrelated application features.

Distributed Feedback

Whenever anyone asks her a question about the isopleth program, she answers by making them, under her direction, write an Explanation Test on her Web site. She instructs her teammates and contacts from other offices to teach each other how to add tests, and not to bother her unless they can't figure out how to write one, or if they break one.

Of course the program has the occasional bug. Two good bug swatters are always available. If a change broke a test unexpectedly, Nancy has the option of discarding the change and rolling the source back to the most recent state where all tests passed.

But when QC reports a bug, Nancy cannot attribute it to any recent edit. QC creates a test scenario to expose the bug, using the program's highest-level interfaces. So the bug could be anywhere within a handful of modules, each full of classes. Nancy's second bug swatter is the hierarchy of tests. She starts by writing a test on the high-level code that reproduces the failing scenario, and instruments it to record the inputs and outputs to the lower level modules.

Then she uses these records to write new tests on each module, and repeats for each class. At each step she adds exploratory test cases to each testing level, from the program level to the module level to the class level.

When she finds the bug, she proves she can write another test—on the buggy method itself—that will only fail if the bug's alive and will only pass if the bug is dead. Then she removes the bug, and ensures that all tests everywhere still pass.

When a traditional project fixes a bug in old code—maybe code that many clients use—the fix risks "domino bugs" in all directions. Nancy carefully inspects the bug's situation, adds more automated tests, and performs more manual tests, to guard against this effect.

The "bug domino theory" never bites her project, because many clients' tests cover that old code indirectly.

> The first functions written, the ones with highest business value, were refactored and reused ever since. Refactoring exposes behavior aspects to new test fronts, and the most reused code has the most indirect tests.
>
> Changing that old, stable code would traditionally cause incredible risk.
>
> Here, changing the oldest code supporting the features with highest business value is lowest risk.

Though her language supports a debugger, the idea of using it doesn't occur to her.

Her design remains at cleanest local maxima. Her time estimates for each new feature are short and reliable. The program contains quite a few more features than originally envisioned. Some "just happened", and some were based on suggestions from users. Nancy takes a week off.

Before an important meeting, a new marketing representative asks Nancy for a stable version to practice a demonstration script on. Nancy takes her to a test server, and shows her how to request any version of the program by date. She shows a Web site with pages playing an animation of a behavior, the date that behavior went online, the date the animation was recorded, and the input and output variables. The marketing representative prints out each page she likes, and arranges them in a narrative order, forming a storyboard.

The marketing representative downloads the latest program version, and then plays with it, using the animations in the test system to learn how to drive everything. The representative follows her storyboard to run the program in a sequence that tells a story.

> Nancy's literate acceptance tests form the Users Manual, and taught a naïve user, in minutes, not weeks, how to sell this application.

Those well-tested numbers, and captured animations of usage scenarios, helped the Marketing Department get a perfect demo script, even though the demonstrator knew almost nothing about the product. During the demo, nobody will notice. The tests *are* the documentation because watching the tests run teaches exactly what the source is doing.

Slack

The day before the demo, Nancy reviews the demo script, and the representative gives her a list of minor usability irritations, and suggestions for new features. Nancy annotates the list, and e-mails it to her project manager, who collaborates with the marketing department to sort the items by business priority. Then, because her schedule is slack (despite it's the day before an important demo), Nancy feels inspired to add one of the features.

The next day, Nancy integrates the new feature, and deploys the latest Production Build to another workstation in the demo room. Nancy tells the marketing representative that the demo script is more important than the new feature, so she doesn't deploy to the demo workstation. After the demo, the marketing representative switches to the spare workstation, and shows the new feature "added this morning".

Nancy always ensures her project can:

- Test relentlessly
- Remain chronically bug-free
- Grow all modules in parallel
- Sustain continuous, distributed review
- Display the GUI, during tests, on command
- Automate tests that "look at" and "operate" the GUI
- Leverage legacy code and data as test fixtures and test resources
- Leverage team behaviors for more test resources
- Make new test samples extremely easy (and fun) to add
- Spontaneously produce a GUI Layer and a Representation Layer
- Keep everyone involved from day one
- Change any code, no matter its age or depth
- Rapidly add one feature at a time.

She can:

- demo at whim
- upgrade any code freely
- answer questions with new tests
- test at any scale
- release any integration
- change direction at need.

Both Sluggo and Nancy frequently reviewed their programs' appearances and responses. But…

The Biggest, Most Important Difference

When engineers interact with programs and source code, each operation is an experiment. Engineers form a mental model of an application, and make a hypothesis, attempting to predict an experiment's outcome. They perform the experiment, compare its results to their prediction, and adjust their mental model.

The biggest difference is what happens to the hypothesis, experiment, and evidence afterwards. Are they thrown away, or added to the application's constraints?

> Sluggo debugs every experiment.
>
> Nancy tests every experiment.
>
> Sluggo's debugging pushed behavior back and forth, without constraints.
>
> Nancy's tests are experiments that tune behavior, then lock it down.
>
> For each big change, Sluggo remembers a handful of experiments, and runs the debugger over them.
>
> For every small upgrade, Nancy makes tiny edits. After each one, she re-performs every previous experiment, instantly.

Over her project's lifecycle, Nancy traded many long hours debugging for each few minutes writing tests. She followed curious but effective inversions of the programming industry's most common practices.

Chapter 16: *Test-First Programming*

Before 150 years ago, doctors believed evil spirits caused disease. Then anesthetics permitted doctors more time to perform surgeries. These doctors typically washed their hands only *after* operating—to clean the blood off. Patients often caught post-operative infections. "The operation was a success but the patient died."

Around the 1850s, one Dr. Ignaz Philippe Semmelweiss experimented by washing his hands *before* helping women give birth, instead of only afterwards. He reported a low mortality rate, but doctors had yet to learn better systems to report and value experiments and statistics, and Semmelweiss used no physical model to advertise the results.

Then Louis Pasteur and Robert Koch spread the meme that diseases were tiny animals eating our bodies from the inside. This physical model explained the hand-washing experiments, backed by early research with microscopes. Doctors such as Scot Joseph Lister added antiseptic solutions to their regimens. However, many doctors still refused to wash or sterilize their instruments—even though these procedures obviously could not harm patients. Doctors in leading and teaching roles, such as Pierre Pachet, the Professor of Physiology at Toulouse, as late as 1872, would declaim that, "Louis Pasteur's theory of germs is ridiculous fiction."

Today we recognize healthy bodies contain millions of tiny creatures living in a balance of cooperation and clumsiness. Doctors decrease the risk of upsetting this balance by scrubbing a long list of things, including the air, before even minor operations, using soaps and procedures that are now required by law.

The programming industry lives in interesting times. While many engineers still believe lack of design planning causes bugs, they have no conceivable reason not to write test code before writing the tested code. That certainly could not hurt the patient. Many believe it helps the patient, and some feel it's the best way to design the patient. Evidence from the field will eventually settle these debates. When Agile companies out-flank slower competitors, detect their leaders' decisions, and respond faster than their troops, that pressure collapses the decision-making process.

Without GUI entanglements, here's the TFP operation. Subsequent chapters that address the GUI will assume familiarity with the cycle, and the first Case Study, SVG Canvas (on page 68), discusses each step of the cycle for a live project.

The TFP Cycle

When you develop, use a **Test Runner**, written in the same language as the production code, that provides some kind of visual feedback at the end of each test run. Either use a GUI-based test runner that displays a **Green Bar** on success or a **Red Bar** on failure, or a console-based test runner that displays "All tests passed" or a cascade of diagnostics, respectively.

Either way, each final result should appear in the same screen location.

Write failing **Developer Tests**, and briefly make them pass. Then refactor to improve design. Run the tests every few edits, ensuring all tested behaviors remain the same.

Engage each action in this algorithm:

- locate the next missing **code ability** you want to add
- **write a test** that will only pass if the ability is there
- run the test and ensure it **fails for the correct reason**
- perform the **smallest change** needed to make the test pass
- when the tests pass and you get a Green Bar, **inspect the design**
- while the design (anywhere) is low quality, **refactor** it
- only after the design is cleaner, **proceed to the next ability**.

That algorithm needs more interpretation. It minimizes reasons not to hit the test button, and maximizes the odds of correctly predicting each test result.

Looking closely into each **bold** item reveals a field of nuances. All are beholden to this algorithm and to the intent of each action. Each action leverages different intents; they often conflict with other actions' intents. Our behaviors during each action differ in opposing, unbalanced ways. Repeated edits with opposing intents anneal code's structure and lubricate its articulations.

A **code ability**, in this context, is the coalface at the bottom of the mine. It's the location in the program where we must add new behavior, or adjust current behavior. Typically, this location is near the bottom of our most recent function. If we can envision one more line to add there, or one more edit to make there, then we must perforce be able to envision the complementing test that will fail without that line or edit.

"**Write a test**" can mean to write a new test case, and progress as far as one assertion. If the new test lines assume facts not in evidence—if, for example, they reference a class or method name that does not exist yet—run the test anyway and predict a compiler diagnostic. This test collects valid information just like any other (and it minimizes all possible reasons *not* to hit that test button). If the test inexplicably passes, you may now understand you were about to write a new class name that conflicted with an existing one.

Alternately, "**write a test**" can mean to take an existing test function, and add new assertions to it. Tests decouple new code from its own mechanics. Many small tests, with as few as one assertion in each, lead to clean code that assumes very little about its environment. However, test cases that address libraries, such as GUI Toolkit libraries with dozens of coherent side effects, are often longer, with many assertions. They reuse the test's scenario, and Get many control properties that hard logic should have Set. GUI Toolkits often optimize screen display time at the expense of preparation time in memory. Our tests, by contrast, will create many GUI objects in memory and then throw them all away without displaying them. This can waste time, so many of our tests will reuse existing objects a few more times before destroying them.

Work on the assertion and the code's structure (but not behavior) until the test **fails for the correct reason**. If it passes, step through and inspect the code to ensure you understand it, and ensure the true reason was indeed correct; then proceed to the next feature.

To **fail for a correct reason**, all other tests must still pass. Tests share code in fixtures, but developing a new test might tweak those. The fixtures must still perform correctly for all other tests before this one may proceed. Often one can write enough of the test that it executes, then

add an assertion to the test, and switch the assertion so it accepts the code's current state. After a successful run, adjust the assertion to now demand the improved behavior.

This book exists because many libraries—most notably GUI Toolkits—make this step completely non-obvious. Research how to apply the next chapter's TFUI Principles to your GUI Toolkit, until your test cases can use fixtures to reliably **fail for the correct reason**, even when failures appear in remote side effects. This one problem is the nut we must crack—how to write failing tests that only improved appearances and responses can pass.

Just before passing the test is the most efficient point in the cycle to test the test, and make sure it accurately enforces the required behavior.

Check that the diagnostic is what you expect. If the test fails for the wrong reason, your mental model of the code's situation may be wrong. If you fix a test failing for the wrong reason, your fix could be wrong. Alternately, the test may have discovered a fault. Then rename the test case to document the fault, use it to fix the fault, and write a new test case that gets you back down to the coalface.

All that work prepares you to make that **small change**. Go ahead and write that line which you have been anxious to get out of your system for the last eight paragraphs.

The **change** is **small** because we live on borrowed time until the Bar turns Green. Correct behavior and happy tests come (just slightly) before design quality. We might pass the test by cloning a method and editing one line in it. If that's the minimum number of edits, do it. Or, re-write a method from scratch, even if it turns out very similar to an existing method. And often the simplest edit naturally extends a pre-existing clean design that won't need refactoring, yet.

Ironically, one should work harder to ensure a test **fails for the correct reason** than one should work to make it pass! See page 102 for this effect. The **change** is **small** because it may do anything, including lying, to pass the test. More tests will force out the lie. Practice this technique, even when you don't know the production code lies.

"**Small**" here means comprehensible, not sparse. Of course complete variable names, proper formatting, etc, are worth typing. But could you manually reverse the **change**, without the Undo button? You should be able to manually find your changes and remove them. Don't keep too many edits in memory until getting back to a Green Bar, and don't tangle the new code up with the old, yet. Only change a few small areas.

New systems often require scratch projects and excessive hacking to learn. Pull this code out of the production code folders, and use it as an example. Then use it as a reference to make each small change in the production codebase.

If the **small change** fails, and if the fault is not obvious and simple, just hit the Undo button and try again. Anything else is preferable to bug hunting, and an ounce of prevention is worth a pound of cure.

After a **small change**, if an assertion fails, inspect its input values. If the production value is better than the reference value, the test is hyperactive. Copy the production value to the reference value and test again. But don't do this too often; see Fuzzy Matches on page 37.

The more mature a module, the more likely the **small change** generates an instant Green Bar. Quoting the eminent Dr. Raymond Stantz, "The light is green; the trap is clean."

Analysis and Design

When we have a Green Bar, we **inspect the design**. Per the **smallest change** principle, the most likely design flaw is physical or conceptual duplication. To teach us to improve things, we spread the definition of "duplication" as wide as possible, beyond mere code cloning. For example, the numbers `4` and `6` may, in context, actually represent `5-1` and `5+1`. Remove optimizations to inspect latent underlying concepts.

The book *Design Patterns* advises, “Abstract things that vary.” This is the reverse way to say, “Merge the behaviors that duplicate.”

> Merging behaviors requires code structures to grow new flexibilities.

To **refactor**, we inspect our code, and try to envision a design with fewer moving parts, less duplication, shorter methods, better identifiers, separated concerns, fewer comments, and deeper abstractions. Start with the code we just changed, and feel free to involve any other code in the project.

If we cannot envision a better design, we can proceed to the next step anyway. Seek minimal edits that will either improve the design or lead to a series of related edits that might lead to an improvement. Between each small edit, run all the tests. If any test fails, hit Undo and start again.

> Don’t refactor in order from hard to easy. Refactor from easy to easy.
> Start by picking the low hanging fruit.

During this step, never change functionality—only design. Refactor anywhere in the program. This recovers the design from the **small change** that only changed a few small areas. Now is the time to reconcile the new feature with any code that could better support it. Frequent refactors irresistibly pull much code, and tests, toward methods containing only 1 to 3 statements or operations.

If a design contains several related problems, don’t begin with the hardest parts. Confronted with a super-long method, don’t pick the midpoint and split it in half. That creates an un-reusable private method. Instead, seek the smallest improvement possible within all its statements. The odds are very good two or more of them duplicate some trivial behavior. Pulling it out into reused support methods might improve flexibility, and (generally) can’t hurt.

The level of cleanness is important here. You may have code quality that formerly would have passed as “good enough”. Or some clever but useless abstraction might enamor you. Snap out of it. The path from cruft to new features is always harder than the path from minimal elegance to new features. Fix the problems, including removing *any* speculative code, while the problems are still small.

Projects without test-first might fall down at this step. Without proactive tests, aggressively removing lines of code adds risk. When code must mature, debug, and stabilize before a cleanup phase, opportunities are lost.

Refactoring into many small delegates, then refactoring to remove duplicated delegation, leads to **Contractive Delegation**. After implementing a complex feature, and after the first round of refactoring, the design might contain many absurd chains of delegation, where A calls B calls C repeatedly. The second refactoring phase isolates and removes the intermediate B methods. This level of abstraction forces special cases to isolate from each other efficiently. Each delegating method accepts its inputs, contracts them by adding this method’s special ingredient, then passes the inputs to the next delegate. New features are very easy to add to code following Contractive Delegation. Code following these patterns can be a little difficult to read. No more 30-line methods with obvious procedures.

> End a refactoring session with Rename Method Refactors. Name things after their new intentions.

If you see duplication, but can't imagine how to improve its design without obfuscating what it does (or can't imagine any way at all), move all the duplicating lines next to each other. This practice forms little tables, with columns that are easy to document and scan.

Early refactors often stabilize a design, enabling new features of the same kind without refactors. Suppose you add two abilities, and they generate similar statements. Merge duplication into an abstraction just as soon as two abilities require it. The odds that future abilities will reuse that abstraction are now very high, because applications by nature present users with lists of similar options. This simple technique prevents code from becoming increasingly disordered and hard to change.

Good designs approach the "Open Closed Principle", which essentially means (in an organic design context), "Reuse me the way others have reused me. Don't reuse me by typing on me, adding '`if`' statements for special cases, etc. Reuse me by adding to one of my lists—metadata, derived classes, clients, plugged-in classes, etc. I am Open for re-use but Closed to edits."

Refactor tests too, but follow slightly different rules. Test cases will duplicate much trivial behavior on purpose, to self-document. Merge such duplication to improve readability, and to create fixtures such as this book recommends.

Grind it 'Till you Find it

Object oriented programming is about messages and behaviors. Objects and classes only support them. Behaviors emerge from the structure of messages, not from properties of any particular class objects. Encapsulation means abstracting the message from its implementation, by both syntax and semantics. While building a complex system, simplifying it by replacing conditional `if` and `switch` statements with messages increases its range of existing behaviors, and its potential new behaviors. They emerge.

When adding behavior, duplicate code on purpose, to then permit Refactors, beginning with Extract Method, to isolate the common cases from the special cases.

> Clone and modify to add features. Then reconcile differences, make things look similar, then the same, then merge them.

To design, we make the **smallest change** that passes a test, then we refactor to improve the design. Put another way, we start with poor design but tested behavior, and move to a better design with tested behavior.

> Frequent refactoring reduces the odds of a big refactor.

Refactoring is not rework, if you generally plan to refactor, and if the design converges on a stable solution supporting many features. But a maturing module might grow until a very different design might work better.

At this time, if developers notice the latent alternative design and decide to express it, the code will undergo a "pop".

Many small refactors get to the other design. The "pop" is not instant. Between each small refactor, all tests must still pass—or developers will undo the refactoring step. This means if the Customer interrupts a long refactoring session to request a release, the developers can ship code that is half one design and half another. The tests don't care, and the Customer Team's business strategists can always decide when to release.

During such a "pop", design quality must go way down, before it can go up again—presumably to higher than where it was before. Design cannot always incrementally improve—this is not a "hill-climbing algorithm"—but you *can* always incrementally *change* the design, toward a potential improvement.

Other books, such as Joshua Kerievsky's *Refactoring to Patterns*, cover how to do this, focusing on situations whose path is not clear. We'll rely on the basics here.

Refactor in small increments so behavior cannot stray too far, and to support Continuous Integration.

Suppose someone interrupts your refactoring session and requests an immediate release, with the latest features. If you have wall-to-wall tests, and if the tests all pass, your very next integration has very high odds of passing QC and shipping to live users. And this means you could ship the product in a half-refactored state. The users need never know the code contains classes half one design and half another.

Finish a refactoring effort with a round of Rename Identifier Refactors, to accurately name everything's roles in the new design, and bond them to the System Metaphor.

Deprecation Refactor

Sometimes a design, or even a feature, needs a complete replacement. Do this by leaving the old system online at the same time as the new one grows, and only when the new can replace the old one take it out. This is an elaboration of the Substitute Algorithm Refactor, but it works at all scales, from entire modules to single variables.

For example, one of this book's Case Studies is MiniRubyWiki. Its chapter, The Web (page 365), thankfully does not report *every* feature and refactor this project has endured. But here's a simple deprecation in action.

Suppose a function assembles a very long string, and returns it:

```
    def formatFile(title, contents)
        form = titleStuff(title)
        form += "\n"

        form += table('width = "100%"') {
                tr{ td('valign="top"'){headerStuff(title)} +
                    td('align="right"'){getBanner()} }
                }
…
        return form
    end
```

That's hard to understand, fragile, and inefficient. The variable `form` has a very long scope, so you must mentally track its status at each juncture. Each `+=` operation concatenates to the end of a growing string, and such growth often reallocates memory.

The code that calls this receives the string and writes it to a stream (such as the `$stdout` file handle, so a CGI host can send it out a TCP/IP connection). We seek code that's both cognitively and physically efficient. This function should take a reference to a stream, so each string segment can simply `.write()` into its destination, without concatenation.

We don't just throw a stream in, change every line, and change all the tests and clients. Every intermediate change would be untestable, causing risk.

Instead, change the smallest testable steps. Start by adding a stream argument that does nothing:

```
    def formatFile(stream, title, contents)
        form = titleStuff(title)
        form += "\n"

        form += table('width = "100%"') {
                tr{ td('valign="top"'){headerStuff(title)} +
                    td('align="right"'){getBanner()}              }
                }
…
        return form
    end
```

That function does not yet use its new stream argument. Now search for every call to `formatFile()`, and pass in a useless stream parameter. Then we expect all tests to pass. The variable `form` is now deprecated—scheduled for retirement.

Next, change the method's external behavior:

```
    def formatFile(stream, title, contents)
        form = titleStuff(title)
        form += "\n"

        form += table('width = "100%"') {
                tr{ td('valign="top"'){headerStuff(title)} +
                    td('align="right"'){getBanner()} }
                }
…
        stream.write(form)
        return
    end
```

This change forces all tests and clients to stop using the return value, and start using the stream they passed in. We minimized changes between tests—the method still uses the `form` variable, even though it's now redundant.

After passing the tests, begin to reduce the deprecated variable's scope:

```
    def formatFile(stream, title, contents)
        stream.write(titleStuff(title))
        stream.write("\n")

        form = table('width = "100%"') {
                tr{ td('valign="top"'){headerStuff(title)} +
                    td('align="right"'){getBanner()} }
                }
…
        stream.write(form)
    end
```

This change leaves all the tests and clients alone, and the variable `form` sees less use. The deprecation is almost finished. However, servant methods such as `table()` still return a string;

they don't take a `stream`. The code still uses half one style and half another, so the entire "refactor party" is not over yet.

That deprecation changed structure first, then external behavior. Changing internal behavior last permits the most opportunities for successful test runs.

During a deprecation, behavior may remain the same, or it might slide. This book informally calls that a "Deprecation Refeaturization"; an example appears on page 206.

The hardware analogy here is an incomplete and partly functional suspension cable bridge. An older stone arch also still firmly supports the road deck. Traffic can still safely cross. But such a design might not appear very beautiful, but it has twice as many tests as before.

Constraints and Contracts

We may add assertions at nearly any time—while refactoring the design, and before **proceed**ing **to the next ability**. Whenever we learn something new, or realize there's something we don't know, we take the opportunity to write new assertions that express this learning, or query the code's abilities. As the TFP cycle operates, and individual abilities add up to small features, we take time to collect information from the code about its current operating parameters and boundary conditions.

Boundary conditions are the limits between defined behavior and regions where bugs might live. Set boundaries for a routine well outside the range you know production code will call it. Research "Design by Contract" to learn good strategies; these roll defined ranges of behaviors up from the lower routines to their caller routines. Within a routine, simplifying its procedure will most often remove discontinuities in its response.

Parameters between these limits now typically cause the code to respond smoothly with linear variations. The odds of bugs occurring between the boundaries are typically lower than elsewhere. For example, today's method that takes `2`, `3` and `5` and returns `10`, `15` and `25`, respectively, tomorrow is unlikely to take `4` and return `301`.

> Like algebraic substitutions reducing an expression, duplication removal forces out special cases.

After creating a function, other functions soon call it. Their tests engage our function too. Our tests cover every logical statement in a program, and they approach covering every path in a program. When features accumulate in order of business value, the code of highest value—written earliest—experiences the highest testing pressure and the most test events for the remaining duration of the project. The cumulative pressure against bugs make them extraordinarily unlikely.

If you are curious, or code does something unexpected, or you receive a bug report, always write a new test. Then use what you learned to improve design, and write more tests of this category. If you treat the situation "this code does not yet have that ability" as a kind of bug, then the TFP cycle is only a specialization of the principle "capture bugs with tests".

Refactoring and Authoring

Programmers working on a GUI module perform three activities. Each has different needs that influence programmer behaviors:

- **Developing**: Improving behavior, using Test-First.
- **Refactoring**: Improving structure, without changing behavior or esthetics.
- **Authoring**: Improving the esthetics and minor behaviors of an application.

> Only switch between activities when all tests and checks pass.

Designing is the process of organizing a program's structure in memory with its behavior in time. Authoring is the process of assembling all the trivial output and input behaviors that don't require hard logic.

Authoring is not GUI usability design. Only develop a GUI toward your current iteration's goals. Never change usability—even at the level of a single keystroke—without review.

This book uses the term "Authoring" as an excuse to bypass TFP when developing minor changes to visual appearances and input event handlers. TFP could conceivably constrain such changes—if the technology permits, if such tests would not interfere with re-authoring, and if the project generally needs such tests. We can't test every pixel, so we draw the line at features with complex behaviors that could affect internal data.

Regulate authoring the same as refactoring—only edit as little as possible, between running all the tests. When authoring, all visual checks and tests must pass. The TFUI Principles will recommend configuring tests to support visual checks, to repeatedly and rapidly display windows, populated by test data.

Noodling Around

Creativity drives coding and designing. If your creative muse compels you to go off alone and write new code (or UML diagrams), without tests or a Pair Programmer, then go write. Introduce the result into a production codebase by trivially re-writing it, this time using TFP and a pair. Proficient TFP can also propel creativity.

Agile methodologies endure accusations that "requiring" engineers to "always" perform certain practices, such as pairing or TFP, represents a belief that "one size fits all". Mature Agile teams often allow engineers to write solo production code. And many projects naturally require long sessions analyzing specifications, and planning architectures, to reduce special risks and promote domain comprehension.

Agility's learning curve, for each project, crests when a team knows which modules cause most risks, and how testing and pairing boost their velocity.

> Feedback adapts a team's implementation of Agile practices to their project's unique topology.

While reviewing and refactoring finished code, one often adds minor statements to improve Sane Subset compliance, and slip in new behaviors here and there. This activity changes behaviors without test-first. On test-free projects, such tweaks are part of the problem. Most bugs happen as old, forgotten code changes to meet new requirements, so that's where test-first

is most important. Frequent test runs make minor tweaks harmless. They rely on “Herd Immunity”.

Every code ability must connect to a live User Story. Put another way, one can add `assert(false)` to generally any code block, and each Customer Test requiring that block will fail. Traditionally, only advanced software engineering methodologies that slowly and carefully planned designs could enable this high level of requirements tracing.

Computer Science vs. Software Engineering

Researching new, complex algorithms is Computer Science; an open-ended quest into the unknown. Software Engineering is a system to efficiently turn the results of research into money, so avoid inefficient, open-ended quests into the unknown.

While TFP can be a very good computer science research tool, its results are excessively sensitive to initial conditions. Programs are simple algorithms—iteration, recursion, etc. Refactoring searches the space of designs, for simple algorithms, to find a good fit. Complex algorithms, after discovery, are published behind simple interfaces, such as the C++ `std::sort` template. When TFP seeks a new complex algorithm, early refactors that obeyed the Simplicity Principles can lead to abstractions that prevent, not enable, a clear and efficient algorithm later on.

Expect to restart such research several times before TFP generates a good algorithm, and support your effort with more and different constraints than Software Engineering typically uses.

Strong the Dark Side Is

TFP’s force can cause problems. Used alone, without checks and balances from code reviews and feature reviews, frequent testing can help add many useless features, providing a very high apparent velocity. Used incompletely, with huge gaps between tests, TFP can reinforce bad code.

Some folks write tests, then write code and “refactor” for a long time, without frequent testing. These behaviors add back the problems that TFP helps a healthy process avoid.

TFP applies our Agile “headlights metaphor” in miniature. Imagine headlights that can only strobe, not shine continuously. Each time you hit the test button, you get a glimpse of your implementation’s road ahead. So to change direction, you must test more often, not less.

Teach your colleagues to write tests on code you understand, and learn to write tests they understand. This learning begins as a team collaborates, at project launch time, to install and use a testing framework.

Test Cases

Here’s a detail of a simple test in a familiar language, without the production code that makes it pass:

```
int main()
{
    Source aSource("a b\nc,  d");
    string
    token = aSource.pullNextToken();  assert("a" == token);
    token = aSource.pullNextToken();  assert("b" == token);
    token = aSource.pullNextToken();  assert("c" == token);
    token = aSource.pullNextToken();  assert("d" == token);
    token = aSource.pullNextToken();  assert(""  == token);
```

```
                                                    //  EOT!
}
```

Passing the test requires objects of type `Source` to parse strings, ignoring spaces and commas. When I wrote the test, following the TFP cycle, I wrote one line at a time, and got it to pass, before adding the next one.

The increment cycle of adding assertions and features often generates tests with a stack of parallel statements, like the above example. If that were production code, one could "roll it up" into a table to make it shorter and easier to understand & extend. But test style guidelines differ from production code's. They trade parsimony for self-documentation.

That test explicitly documents a boundary condition—`End Of Text` returns blank strings `""`.

Tests document source code. Refactors that reduce production code's ability to be documented by tests are unlikely. Self-documentation is an emergent property of tests, but engineers must also apply their intent. They must read and review tests for legibility, and write extra tests with this primary purpose.

Tests help new programmers learn how to call functions. When you learn a new function, you typically don't just read its prose documentation. You search for a compilable example of it in action. Projects using TFP generate copious self-documentation as a by-product of efficient development.

Large projects should not pack all their assertions into `main()`. Split assertions up into functions called Test Cases. Each is essentially one "paragraph" of testage, following the "Triple A" format:

- **Assemble** the sample object and data: `Source aSource("a b\nc,  d");`
- **Activate** the target feature: `token = aSource.pullNextToken();`
- **Assert** the results: `assert("a" == token);`

Test code must appear "obvious", so each case tells a little story about the production code it tests. Behavior duplicated between tests should merge into common methods. More than one Test Case sharing the same private test fixtures form a **Test Suite**—typically as methods of a Test Suite class. Some test fixtures are important (especially the kinds this book requests), so they should merge into test utility modules that support many Test Suites.

This book uses "Test Rig" to casually refer to mechanisms supporting test code. Other books cover the outstanding *Unit category of rigs in great detail; we will only pay attention to their must-have features. They all permit **Incremental Testing**, so the Test Button doesn't waste time testing distant modules, and they all offer systems to set objects up and tear them down before and after each case.

Chapter 10, *NanoCppUnit*, builds a test rig for C++ from scratch, starting on page 161. Most cases will set up and tear down a dialog.

Test Isolation

Test Cases must not depend on each other, so they could execute in any order. Some Test Suites execute their cases in alphabetical order, some in compiled order. If one Test Case wrote a file, for example, and programmers knew which Test Case comes next, they must not exploit this information to reuse that file. Use your Test Suite architecture correctly to assist "Test Isolation". If two or more cases need that file, they must share a common method that creates it.

Test Fixtures

The product of Extract Method Refactor between Test Cases is a Test Fixture. Put another way, a Fixture is any code (function, class, or method) shared between many real or potential Test Cases.

Fixtures make new similar tests easier to write. When named according to their intent, they might make cases easier to read. When you learn something about your environment, invest that knowledge in a new Fixture.

The TFUI Principles are common goals for Test Fixtures that adapt your GUI Toolkit to your project. The Case Studies reveal fixtures to assist predicting a window's esthetics, and to assist simulating input events. Most Fixtures obey the Simple Design practice; you don't code them until you need them.

Some TFUI Test Fixtures exist only to be used temporarily. Write these proactively, so they are ready when you need them. All our Case Studies will reveal how learning to write such temporary fixtures assists writing important permanent fixtures.

The two most common Test Fixtures are `setUp()` and `tearDown()`. Test Suites call these before and after calling each case, respectively, following the Execute Around Pattern. `setUp()` will initialize Test Suite member variables that each of its Cases can use, and `tearDown()` will clean up any after-effects of each case.

The pseudo-code to run each Test Case in a Suite is:

- For each Case in a Suite
  - Construct a Test Suite object
  - Call `setUp()`
  - Call the Test Case
    - Assemble
    - Act
    - Assert
  - Call `tearDown()`
  - Destroy the Test Suite object

If the Test Suite has data members, they destroy after each case runs.

To ensure Test Isolation, create target objects as data members of the Test Suite object, or as local variables inside a Test Case.

Programs need two kinds of assertions, terminal and diagnostic. If a C or C++ program absolutely must abort its execution, or something very bad will happen, it can call the Standard Library `assert()` function. This prints an error message and then turns off the program. The operating system may clean up the program's resources, but external side effects, such as databases or hardware, may strand in an intermediate state.

Tests use diagnostic assertions. Some test rigs halt the failing case, then allow others to run. Test rigs should permit programmers to manage these details as they see fit.

If production code throws an exception, the Test Runner will use your language's keywords, such as `catch` or `finally`, to `ensure` that `tearDown()` always calls. Applications using a database, for example, often populate it with generic (and possibly mildly random) test data, then test by changing tables and records. Putting a "rollback" call in the `tearDown()` will back-out any edits, leaving the database ready for the next test case—even if another workstation runs it.

This book tends to use—but doesn't require—`setUp()` fixtures that create windows for test, and `tearDown()` fixtures that destroy them.

Some Test Runners provide many elaborate features, such as listing each failing assertion. Test Rigs in general need that ability, but Test-First Programming does not strictly require it. A single failing assertion is cause to suspect the most recent edit.

Write fixtures to merge common test behavior from any set of the Assemble, Act, and Assert steps. Here's a suite that calls a fixture (`test_a_b_d()`) to Act and Assert, but not Assemble. The fixture verifies that whatever input we Assemble, the parser shall pull the tokens `a`, `b` and `d`, but never `c` because the input always comments it out, one way or another:

```
    struct TestTokens:  TestCase
    {

    void test_a_b_d(string input)
        {
        Source aSource(input);
        string

        token = aSource.pullNextToken();
        CPPUNIT_ASSERT_EQUAL("a", token);

        token = aSource.pullNextToken();
        CPPUNIT_ASSERT_EQUAL("b", token);

     // token = aSource.pullNextToken();
     // CPPUNIT_ASSERT_EQUAL("c", token);

        token = aSource.pullNextToken();
        CPPUNIT_ASSERT_EQUAL("d", token);

        token = aSource.pullNextToken();
        CPPUNIT_ASSERT_EQUAL("", token);  //  EOT!
        }

    };

    TEST_(TestTokens, elideComments)
    {
        test_a_b_d("a b\n //c\n  d");
        test_a_b_d("a b\n // c \"neither\" \n  d");
        test_a_b_d("//\na b\n // c \"neither\" \n  d//");
…
        test_a_b_d(" // \na b\n // c \"neither\" \n  d//");
    }


    TEST_(TestTokens, elideStreamComments)
    {
        test_a_b_d("a b\n /*c*/\n  d");
        test_a_b_d("a b\n /* c \"neither\" */\n  d");
…
        test_a_b_d("//c\na b\n // c \"neither\" \n  d/* */");
    }
```

The … mark replaces a great number of similar lines. This book uses ellipses to avoid boring the reader more than the compiler. Those tests cover many related situations.

The books *Test-First Programming: A Practical Guide*, by Dave Astels, and *JUnit Recipes: Practical Methods for Programmer Testing* by J.B. Rainsberger, address a wide range

of testing patterns. They show how to test your code in relationship to many common test-hostile architectures.

Testing Extreme Programming, by Lisa Crispin and Tip House, integrates these patterns with a project's lifecycle.

> All GUI engineers should carefully study Chapter 19 of *Testing Extreme Programming*.

Test Collector pattern

The `TEST_()` macro (defined in the *NanoCppUnit* Case Study, around page 184) builds a list of Test Cases, each inheriting the fixture `TestTokens`. A Test Runner can find every case defined with `TEST_()`, and run them all. Without the `TEST_()` macro, we would need to build a list of cases. Some C++ test runners require this style, from another parsing project:

```
    void test_translatePoundSign()
        {
…
        }

    void test_translateColon()
        {
…
        }

    void test_translateLowerE ()
        {
…
        }

    CPPUNIT_TEST_SUITE(CMainFrame_Tests);
        CPPUNIT_TEST(test_translateColon);
        CPPUNIT_TEST(test_translatePoundSign);
        CPPUNIT_TEST(test_translateLowerE);
    CPPUNIT_TEST_SUITE_END();
```

That's redundant typing, so our computer should help us. Some Test Suites use their languages' **Reflection** system to find all the methods starting with `test_` automagically. C++ efficiently comes closest with the `TEST_()` macro.

Test Resources

When a test fixture, such as `test_a_b_d()`, abstracts from its input data, such as `"a b\n //c\n d"`, this book call input data records **Test Resources**. As they grow useful, they can move into a database, with a simple user interface. That permits anyone who thinks of sample data to enter it, without programming, and see their results. Acceptance Test Frameworks store "test resource databases", forming part of a project's Customer Tests.

Our last project, The Web, creates an environment to store a database of these resources. See page 404 for the last few steps preparing a Web browser to host the data, then sending them through their test fixture.

Incremental Testing

A developer changing a module must run at least all the developer tests that helped create that module, plus any other fast tests that constrain that module. Your editor and its One Test Button should collude with your tests to efficiently run the batch that covers your recent edits. Tests occupy a hierarchy, from fast batches addressing modules, to slow batches of batches constraining entire applications.

Our single test button must respond rapidly. This is not churlish impatience, it's pragmatic. Engineers make small changes and test each one, without worrying about things unrelated to the current task. Delays and discontinuities derail their train of thought.

To test incrementally, maintain a rough one-to-one alignment between test suites and modules of production code. Refactoring may blur this alignment. Useful methods born in one module may migrate to another, away from their tests. When refactoring old and popular methods, run larger batches, to test their client modules.

Test suites should partition pragmatically, not slavishly mirror program classes. Test cases fall together into test suites based on which fixtures they use—not what class or module they claim to test.

To incrementally test segments of an application, put one module in each folder, with its test cases. Tests on any source in this module should cover features well enough to catch mistakes, and should run fast enough to maintain Flow. Tests should compile and run in less than a few seconds.

> "40 seconds? But I want it now!"—Homer Simpson

A good rule of thumb: If a folder full of tests and production code takes too long to compile and test, split the folder into two.

A top-level script checks out, optionally cleans, builds, and tests every possible thing in an entire system. This script, often called "`AllTests`", is the keystone of Continuous Integration. Some languages, such as C++, test many details at compile time, so such a script should also create a Production Build.

When the script finishes, it only prints "All tests passed" if every compile and test in every module in every sub-folder passes.

If the top-level tests take too long to assist Continuous Integration, take time to find the bottlenecks & slow tests, and tune & prune the system. If this effort fails, it might be time to split the project (and team) into two.

Test Hyperactivity

Many TFP tests are hyperactive—they constrain more than strictly needed to boost user productivity. TFUI tests will leverage hyperactivity when a more accurate test is too expensive and complex to write. Passing module-level tests and then failing program-level tests often exposes hyperactivity. Hyperactive tests should migrate down, into the class-level tests. This ensures that sloppy refactors get detected as early as their sloppy edits.

Integration Tests

Some tests will be slow by nature. Include them in your top-level batch—the one often called "`AllTests`". Run all these tests before and after integrating. See Continuous Integration, on page 50, for the complete schedule.

The batch of all tests must bubble up failure conditions from any called batch. If you start an integration test run, wander away, come back, and see a console with "All tests passed" at the bottom, that message must accurately refer to all tests in your shop, not all tests in the most recently run batch.

If a module's local tests pass, but other tests fail, respond by reproducing the failure within a narrower test within that module's test batch.

Continuous Testing

The most ambitious research in TFP focuses on editors that chronically run tests, at every relevant juncture, providing Zero Button Testing. An editor might, for example, draw pink lines under failing test cases, and under the production code they target, much the same way word processors mark words not found in their dictionaries. These techniques require more study to fit into our canonic TFP operation.

Agile literature speaks of many fairly aggressive programming activities—Test Fixtures, Merciless Refactoring, Incremental Testing, Relentless Testing, Continuous Integration, Frequent Releases, etc. They all require a significant effort learning, encoding into a project, and configuring environments & build scripts.

These only burden a project as it starts. When tests grow & share useful fixtures, code grows flexible, and build scripts robustly enable a team's practices, you will write new complex fixtures less often. You will refactor in tiny nudges, and rely on your environment to take care of administrative details. A mature TFP development cycle consists mostly of re-using existing fixtures, adding 2- or 3-line tests, trivially passing them, and occasional small refactors. The feeling of propulsion and momentum is unprecedented in software engineering.

Given a hard problem, TFP generates a solution, with a firm design hosting flexible bug-free features, sooner than expected. *This* is why Agile processes fear too much abstract design planning. Complexity that's too easy to add becomes hard to remove.

Conclusion

Over the past decade, the TFP operation has turned many projects into 99% inspiration and 1% perspiration. But if a Test Case calls GUI code, a window pops up, breaking the cycle, your train of thought, and often the test rig. Early development with GUI Toolkits must write a special category of test fixtures, to defeat each of a list of common problems, including problems caused by excessive test fixtures. And developers must right-size this effort, without burning up time, or taking shortcuts.

Chapter 17: *Exercises*

Based on the Case Studies' online code, use Test-First Programming (or authoring!) to add these enhancements and features (and stomp a few insects):

Any Case Study

1. Re-write a Case Study from scratch using your favorite language and tools.
2. Research a Case Study project's competition, and add the highest business value feature which we don't support yet.
3. Give some assertions comment strings. Make them fail and ensure they adequately describe their situation. If necessary, upgrade the assertion system to transport comment strings. Provide a "verbose" Test Mode that prints complete commentary for each assertion.
4. Add comments, then replace as many of them as possible with better self-documentation, assertions, and test cases.
5. Localize the project to a challenging locale.
6. Connect the cases to a fixture. Connect that to an Acceptance Test Framework. Send resources in and collect test output.
7. Instrument cases with ImageMagick, or an alternative, to record images of its tested window.
8. Pick 3 assignments within one Case Study, estimate how long you expect to finish them, finish them, compare your estimate to your actual time, and repeat.

SVG Canvas

9. Build this Case Study again, from scratch, with any non-Tk canvas control.
10. Extend Ruby::Unit to reflect each assertion's source code into its diagnostic message. `assert_equal(expect, x * 9)` should output "`expect(24) != x * 9(18)`".
11. Interpret more styles correctly, including fonts.
12. Extend the style system, using that old trick, special comments inside another language. Let special comments inside the DOT notation generate animated "pipes" between nodes. Apply this to the gas jungle of a chemical vapor deposition furnace. Animate the pipes by applying a texture, then add a window timer that moves its origin coordinates. Ensure the pipes animate movement in the correct direction between nodes. Trigger the animations when the real furnace turns on a flow in one of its real pipes.
13. Incorporate more GraphViz sample files into test cases. Put a collection of DOT files into a folder, and write a case that traverses this folder, displaying each one. Convert any display bugs into new cases that reproduce the bug directly, with its own DOT notation. Then kill the bug.
14. Add scroll bars to help display huge graphs within a small viewport.
15. Add the ability to edit shapes using menus, keystrokes, etc.

16. Add the ability to save and load SVG to a file.
17. Incrementally add an alternate SVG generator. Ensure it coexists with `dot`.
18. Use GraphViz as a "Geometry Manager" for a non-graph project. Censor the lines, and use it to lay controls out esthetically.
19. Integrate DotML, found here: `http://www.martin-loetzsch.de/DOTML/`
20. Investigate Turtle Graphics, and create a 2-dimensional fractal generator based on simple evaluated Ruby expressions (see `http://flea.sourceforge.net` for the 3-dimensional version of this system). Store the fractals in SVG, and display & tune them in TkCanvas. Add genetic programming techniques to compose new fractals from successful ones.
21. Add a `main()` function that takes the name of a DOT file on a command line, and displays it.
22. Add a top-level window, and place the TkCanvas in this with a TkText control. Put the currently displayed DOT file into the TkText control. Syntax-highlight the DOT notation. Add code-completion context menus to the TkText control. Synchronize the TkText cursor's location, by context, with the selection emphasis location inside the TkCanvas. Permit edits to either the TkCanvas or the TkText control to update the other.

Family Tree

23. Find a large family's descent graph, and enter all of it.
24. Refactor the final version to remove duplicated definitions of behavior. But take care—many of them are inside nested event handler blocks. Beef up the test cases until they are shown to catch any mistake you deliberately attempt.
25. Click on a node. It selects. Now click directly on the text inside a node. Its node does not select. Bind event handlers to `TkcText` items to fix this. Refactor the code to minimize redundant behaviors.
26. Select a node. Tap <F2>. An edit field appears over this node. However, no tests constrain the field's location. Pretend you found a bug—an edit field popped up in the wrong place. Write cases that constrain edit field behaviors.
27. Find a reason to use any of the arguments that `click()` passes to its bound event handler.
28. Arrow keys should navigate the current selection. <Left>, for example, should navigate to the closest node in a quadrant centered on a line going left. Tip: Use the Pythagorean Theorem.
29. Add a Logic Layer that detects impossible family situations, and prevents edits that would cause them.
30. Add a Context Menu to each node, and invoke it from <Shift+F10> and from the Context Menu button on your keyboard. Anything a mousestroke can do, an item on this menu can do. Include submenus "Add Parent" and "Add Child"; each displays a list of valid candidates.
31. Let <Alt+Arrow> extend a dotted arrow to a potential child node.
32. Let <Shift+Arrow>, or a mouse dragging a "Pick Aperture", select a range of nodes. Permit a context menu to do something to all of them simultaneously.
33. Let nodes display biographical information in popups.
34. Let nodes contain little photographs of family members.
35. Write a DOT file that draws a canvas that breaks a test patterned after `test_writeDOTfile()`. For example, `dot` likes the file's syntax, the file obeys the Family

Tree Format, and the file doesn't look like a family tree. For extra credit, drive the Family Tree application's user interface to produce this file.

36. Write `main()`, and provide "Load" and "Save" buttons in the frame around the canvas.
37. Add "Delete" to the context menu.
38. After dragging an arrow from a parent to a child, the code refreshes. The refresh does not restore the selection emphasis to the currently selected node. Fix this, if the node still exists.
39. Add hierarchical Undo.
40. Add scrollbars, to view large families within small frames. Ensure the selection emphasis is always visible. Center it if it goes outside a viewport.
41. Add a horizontal layout option, using the DOT `rankdir` command.

NanoCppUnit

42. Merge duplication among the various rampant `#define` macros. Tip: Use Extract Method Refactor to create utility methods which macros call.
43. Merge duplication by deriving an `iostream` class that works like `cout`, but emits contents to both the console and `OutputDebugString()`.
44. Add the optional ability to output a list of failing tests, instead of crashing on each one.
45. Add a `CPPUNIT_MATCH()` that matches regular expressions.
46. Add an exception handler to the test runner's working loop, so tests after a fault can recover and run.
47. Convert the program in Release Mode to link using the `/SUBSYSTEM:WINDOWS` flag.

Model View Controller

48. Write the project using VS6 and fixed-length strings. For extra credit, use variable-length strings.
49. For more extra credit, write the project with MFC, or raw MS Windows API calls in C.
50. Write the project using CppUnit, or another test runner.
51. Add a Logic Layer that uses customer names and addresses for some useful purpose, maybe using some alternative persistence format.
52. Split the project into separate `.cpp` and `.h` files, so each translation unit sees the minimum volume of imported identifiers.
53. Give each nascent module a separate test suite.
54. Add all the states to the State field. Fix the data layer so each address does not duplicate all these states.
55. Internationalize the application by adding a country field. Display all countries (including a few their neighbors refuse to recognize) as localized names, and store them as 2-letter ISO 3166 codes.
56. Numeric edit fields should reject alphabetical characters. Ensure this feature works correctly in all locales. Use `SendMessage()` at the keystroke level to test that rejected characters make a warning beep.
57. Minimize the number of string types this program uses in interfaces.

Broadband Feedback

58. Improve `ProjectDlg`'s ability to prevent users from losing their data, or unintentionally overwriting it. Add the features suggested on page **Error! Bookmark not defined.**.
59. Keystrokes into the list box should select customers by last name. As the user types more keystrokes, they should accumulate, until typing enough of a name selects that one customer out of a long list. Visually reflect the keystrokes by partially highlighting the text of the first occurrence of a sought last name. Use tests on MetaFiles to generate this behavior.
60. Typing on the name fields should incrementally select this name in the list box. Typing a new name creates a new record; typing part of the same name and tapping <Enter> selects an existing record.
61. Make the progress bar display a rainbow as it progresses. Use tests on MetaFiles to constrain this "owner drawn" behavior. Merge all fixtures that use MetaFiles, to make them more proactive. Write a new complex graphical behavior, such as a background image on the entire dialog box, using strict test-first and these fixtures.
62. Add a new locale:
    - a frequently localized language, such as Spanish or Japanese
    - your most familiar non-English language
    - an ideographic script, such as Chinese
    - a right-to-left script, such as Arabic or Hebrew
    - a common but rarely localized language, such as Wolof
    - a dead language, such as a Sumerian glossary in a Cuneiform script
    - the ancestor of your language
    - an imaginary language, like Klingon, Sindarin, or your inner spirits'.
63. Ensure new locales are plug-and-play, and put them in dynamically linked modules.
64. Write tests that take pictures of each locale.
65. Write a fixture to sanity-check the relationship between a string and a font, using a function different from `ScriptGetCMap()`.
66. Write an acceptance test that collects every string and pushes it into grammar checkers and spelling checkers for its locale.
67. Use either the presented Abstract Test framework, or a better one, to run all tests against all locales.
68. Replace the `Sleep()` call inside the progress bar simulation with a function that actually does something. Make the function correlate many input records together. Note this function must be fully event-driven.
69. Before the previous exercise, fork the project into two separate versions. Leave one the same, and in the other replace the window timer with a thread. Replace `Sleep()` with a function that actually does something. Observe which version becomes fragile.
70. Write a test fixture that records an animation of the progress bar progressing, for each locale, and uploads all animations to a Web server.
71. Record an animation of a fast process, such as data entry operating character by character, and instruct ImageMagick to compile an animated GIF with a delay for each cell, so observers and see what's going on.
72. Add error handling, recovery, and reporting to the `capture()` system.
73. Separate the slow test fixtures into a separately compiled program. ImageMagick is slow by nature. Alternately, configure ImageMagick to store files without compression.

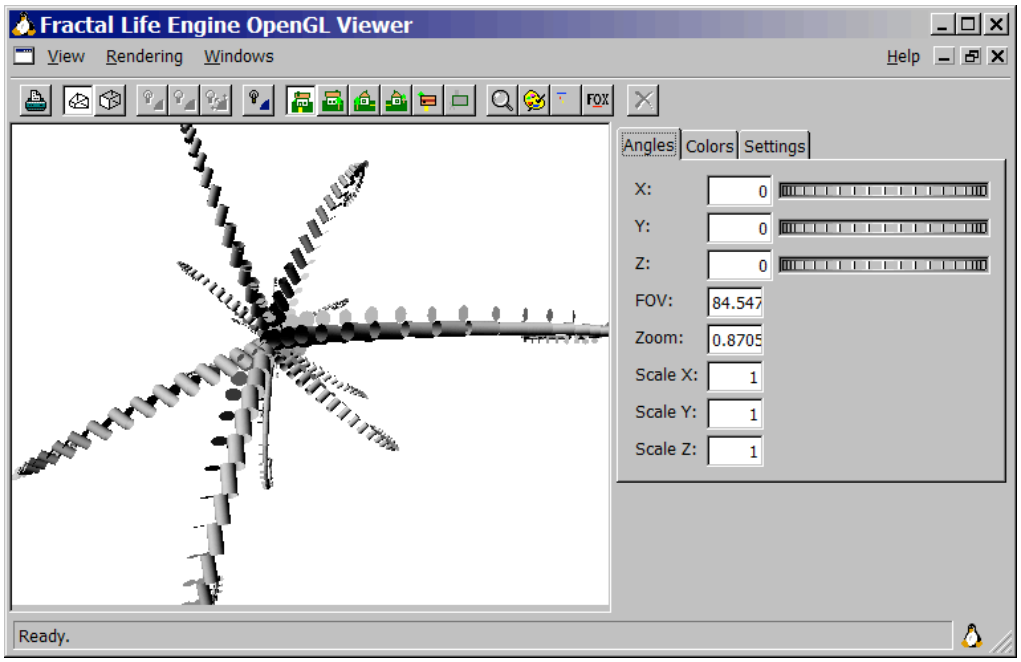
74. Accept command lines containing XPath expressions that return more than one node. Permit a missing XPath expression to test all nodes.
75. Trigger `revealFor()` based on the name of the current computer, not the user.
76. Write XML test resources that operate each control by name.
77. Write a batch file that checks-out and builds the test executable, if needed. Configure an Acceptance Test Framework to trigger this batch file before running the test cases.

Embedded GNU-C++ Mophun™ Games

78. Point a toy in the direction a character is moving.
79. Add a Logic Layer, with representations of characters' health points and the effects of toys.
80. Add more toys & monsters.
81. Let monsters play with toys too.
82. Add a random map of walls. Scroll them when a character goes out of range.
83. Conditionally compile abilities for more advanced cell phones.
84. Automate complete play scenarios, where the hero defeats multiple monsters within a non-random map. Ensure code changes that affect play balance trigger these tests to fail, by killing the wrong characters.
85. Get the 3D version of Mophun and put critters into an isomorphic view.
86. Write test fixtures that locate objects via high-level locations, not pixel offsets.
87. Write a test fixture that records scenarios as animated GIF files, and uploads these to a Web server.

Fractal Life Engine

88. Get the `pigment` working.
89. Replace the lines with `gluCylinder()`, or hexagonal prisms.
90. Clicking on an OpenGL primitive selects the Flea source command that created it.
91. Add all the user interface controls that "`fleaFX`" borrowed from "`FXRuby/glviewer.rbw`":



92. Add fog, as a depth cue.
93. Rendering is slow. Flea should time-slice rendering, so we can add a window timer to pump it or cancel it.
94. The `glFrustum()` logic is brain-dead. It needs an algorithm to locate a target object.
95. The editor is primitive. To save a fractal, you must copy its source out and save it in a "`*.flea`" file. The editor should be replaced with either a real programmer's editor, or should upgrade to reflect awareness of Flea's behaviors.

The Web

This Case Study built no single application from scratch. The living MiniRubyWiki project has its own do-list, available online at `http://minirubywiki.rubyforge.org`.

These suggestions may apply to any Web project, any Wiki, and any test browser:

96. Extend HttpUnit, or a similar test rig, to download a swatch of daily online comic strips.
97. Write a `reveal()` fixture, at the HTML or HTTP level, which writes a temporary file and raises your Web browser to display it. Ensure any included graphics and Cascading Style Sheets appear correctly.
98. Write a test fixture that collects animated visual output. Configure a test browser so clicking on an animating GIF leads to a page with each cell in the animation decomposed into separate images.
99. Time-stamp XML test cases with the date the test went online, and the date it last ran.
100. Link FIT-style test resources to MRW, and MRW-style test fixtures to FIT resources.
101. Configure a test case to record images of a Web browser hitting a Web page.
102. Configure Apache, on WinNT-derived platforms, to evaluate test fixtures outside its service restrictions.

103. Adjust a test browser's behavior so XML input that never uses its node contents does not display a big empty `<textarea>`.
104. Support arbitrarily nested test cases (per page 395).
105. Adjust a test browser's behavior so tabular XML input forms tables with redundant attribute labels in headers, and columns below them with editable fields.
106. Add to MRW's Logic Layer a table of potential fixtures in a given system, each with a corresponding blank XML template. Test authors can select from the list and build a new testing page using a page with a <form> which is itself generated from XML and a fixture.
107. Install MRW on a PDA, such as a Sharp Zaurus, that supports WiFi, Linux, and Ruby. Trigger remote test fixtures from this environment.
108. Configure WebInject and MRW to transclude WebInject's optional HTML output that describes its test activities.
109. Configure WebInject to trigger the MRW page that launches WebInject's MRW test that triggers the WebInject page to…

Glossary

Aggregate Control—a control assembled from a set of previously invented ones. The most familiar example is a Combo Box. This combines an edit field and a list box. Each control shares input events with its colleagues to efficiently reflect user intent

Bogus—lacking veracity or merit

Clone and Modify—if your boss pays per line (directly or indirectly), this technique is your friend. But removing that duplication in a different way delivers Contractive Delegation.

Code ability—smaller than a code feature

Code and Fix—writing code without following a planned design, or writing tests to permit refactoring, followed by debugging for a very long time

Codebase—all of Production Code, Test Code, and support scripts.

Common Code Ownership—you can't tell who wrote what code. Anyone may change any part of the Codebase at need. Continuous Integration prevents conflicts

Common Style Guide—Arguing about technical rules is good for the code. Arguing about esthetic questions is bad for a project, especially when the code is used to argue. Code should look like one (very smart) person wrote it—not like a ransom note. "Esthetic style guidelines" are rules that a pretty printer such as GNU's `indent` could enforce. They could change the placement `{}` braces, for example, without changing the program's design.

If new source contains various different styles, suspect lapses in other aspects of your process, such as frequently changing pair partners, or Shared Code Ownership.

Compilable Comments—Crufty source code statements whose only value is self-documentation. They could be refactored away.

> I frequently see novices creating classes that exist, but don't even have any methods or data members. No code could possibly be written to use the class (except as a base for cross-casting, which is also not done), so it goes without being used. It's "useless".
>
> Obviously someone thought that the classes provided meaningful context. This is why I call them "compilable comments". They could just as easily have been actual comments. –Tim Ottinger (on news:comp.object)

Continuous Integration—When many engineers work on the same project, they must share a common baseline of the code in a database, called a version control system. However, this means if two engineers simultaneously change the same module, they risk changing the same line at the same time. Traditional lifecycles lock entire modules, granting individual engineers exclusive access to them. But delaying integration compounds the cost. If changes were darts thrown at a map, engineers' odds of writing conflicting changes would parallel the odds of two darts hitting the same country. However, if each engineer pulls their dart out of the map very soon, the odds of a coincidence go down. Integrate each time your code improves in a way, however slight, that your colleagues should share.

TODO point out some CI examples.

Contractive Delegation—well-factored code often has many small functions. If each adds value, and doesn't just pass the buck, then what do they all do? Typically, they contract their input by making it more specific. Then they delegate these specific data to a delegatee. For example, this function takes a `fileName`, augments it with the current document, then passes them both into the File system delegatee:

```
def save(fileName)
    string = get_chars(0, get_length())
    File.open(fileName, "w").write(string)
end
```

Every short method solves one part of the problem, and delegates the remaining part of the problem to other methods. Every method transforms its input before passing it to other methods, and/or transforms the output of other methods before returning to the caller.

Control Flow—the location of the program's execution context.

Customer Team—the role that owns the business priority of each User Story, and that authors Customer Tests.

Customer Test—a literate test with a custom user interface, revealing the Customer Team's project specifications, in a self-documenting format. Write and pass at least one for each User Story. Reusable custom user interfaces for Customers Tests form Acceptance Test Frameworks.

Cruft—a word merging "crust" and "fluff". Source code that adds more risk than value.

Custom Control—the biggest challenge in the book, so avoid mostly.

Design Smells—feelings or attitudes about code quality, which may or may not lead to rational explanations. Refactor anyway and see if the design gets better—but only if you have tests.

Developer Test—a test whose unexpected failure implicates only the most recent edit. Contrast with Unit Tests, which require Mock Objects to isolate their faults to one unit. If we test between each edit, we need fewer mocks. Real Production Code objects can assist the tested code. This lets us write informal and useful tests more rapidly.

Display Bug—one that affects the user's experience but not the persistent data or side-effects.

Dynamic Typing—contrast with Static Typing. Languages like Smalltalk, Python or Objective C bind method calls at run-time.

```
aObject.method(argument);
```

That line will evaluate for any `aObject` that has a `.method(argument)` to call. Think of it like the compiler just reads "`method`" as a text string, and looks for that string in the given object's member list.

Esthetic Style Guideline—for example, I put `()` after most method calls, even though Ruby doesn't require them. It's an "esthetic" guideline because you could change it without changing the meaning of any statement. Contrast with a **Technical Style Guideline**.

Event Driven—when programs set up an Object Model in memory, then use the Observer Pattern to await notifications from other processes. Each notification changes the state of Object Model. Contrast with "procedural" programs, which use the current location of Control Flow to store their state.

Extract Algorithm Refactor—convert legacy code into test fixtures and Test Resources, and use these to generate new code.

Extreme Programming—a set of distinct, primal, aggressive, and useful Best Practices that often fit together into a rapid software engineering lifecycle tuned to prevent stress and risk.

Frequent Releases—regularly build a version and push it out of the lab, typically every week. Whether users should receive it becomes a business decision.

Graphical User Interface—a module that presents an arbitrarily complex visual appearance, and enables mouse and keyboard interaction.

Green Bar—the reward when you have incrementally improved a program, using TFP and a test rig that displays a graphical test runner with a progress bar that turns green.

GUI Layer—A "Layer" is two Façade patterns. This Layer presents one Façade toward the GUI Toolkit, and the other toward the Representation layer. Some literature calls this the "Presentation Layer".

Incremental Testing—the ability to run one module's tests in isolation from the entire application's tests.

Integrated Development Environment—an editor that bonds with a compiler's features, providing syntax highlighting, project management, debugging, form painting, etc.

Integration—publish your code changes to others' workstations, and others' changes to yours.

Layer—a module with two or more Façade Pattern fronts, one for the layer above, another for below.

Logic Layer—the layers of a project that store and process data, independent from their GUI.

Message Pump—it's `mainloop()`, the function that blocks until a GUI receives input from the OS or the user, then dispatches some messages to bound GUI Layer code.

Model View Controller—a triad of three modules linked by the Observer Pattern, all residing within the Representation Layer. The View drives a presentation within the GUI Layer, and elements within the View observe the Model. Elements within the Controller observe the View and Model, and elements within the Model observe the Controller. The Model fronts data objects within the Logic Layer.

This pattern decouples changes to how data are manipulated from how they are displayed or stored, while unifying the code in each component.

Alternative versions of this pattern appear in many architectures, and with various layer affiliations. Simplifications include leaving the View inside the GUI Layer, Observing an entire component instead of elements within it, or using direct method invocations instead of Observation messages. Complications include substituting any element of the three components with a different component inheriting the same interface.

Mock Object—a test fixture passed into the tested code as a placeholder for a Production Code object. Mock methods log what the calling code does to them, and return prefabricated values. Use a Mock to test…

- what time is it
- releasing resources
- pseudo-random numbers
- without excessive side effects (such as `MessageBox()`)
- expensive or slow hardware
- modules that depend on modules that don't exist yet
- 3rd party modules that can't Get everything they Set
- threads, semaphores, and other asynchronicities
- responses to failing system calls.

Muscle Memory—pseudoscientific term referring to memorized sequences of actions. The brain thinks of one action and the body performs many. This effect often covers gaps in GUI usability.

Observer Pattern—an advanced form of callback. If `ObjectA` needs to wait until `ObjectB` has made a decision, then `ObjectA` could use a loop statement. This is both mechanically and conceptually inefficient. ObjectA should instead pass a handle to a method to ObjectB, which should make its decision and then call the method. All GUI Toolkits use the ODP for their event dispatch systems.

Organic Design—to construct a large complex design, write one feature, get it working, and simplify its design. Then add another feature, and simplify the design so everything merges. Repeat until a minimal set of code supports many features. If the design were diagrammed at each step, its model would appear to grow like a plant.

Pair Programming—two engineers with one workstation and one task. Take turns typing, and switch pairs frequently.

Planning Game—a regular meeting, typically every two weeks, to assess features completed in the previous iteration, and select the features to implement in the next iteration.

Production Code—the source that compiles into end product—distinct from Test Code

Production Build—to Frequently Release, the build scripts must create a complete application, with its installation and delivery systems, without any Test Code.

Programming by Intention—sketch client code first, to force server code to present interfaces most convenient to clients.

Quality Control—a role within the Customer Team that authors Customer Tests, adds Unit Tests, and collects metrics.

Some sources call this role "Quality Assurance". Agile projects grant professional testers the responsibility and authority to control quality, not just measure it.

Refactoring—changing the design of existing code without affecting its behavior. Refactor in the smallest steps you can think of, and run tests between each test.

Refactor Mercilessly—change any code in any module for any reason. This permits Simple Designs to morph as a project supports new features, but only if tests and continuous review help refactors resist new bugs.

Reflection—a programming language's ability to support queries about a program's own internal structures.

Regular Expressions—the most common implementation of a minimal language to parse and manipulate text strings.

Representation Layer—a layer that changes one representation into another. Between the Logic Layer and the GUI Layer, it could change normalized database tables into wide data views in a format a user would recognize. An engineer familiar with the user interface would see all the same features in the Representation Layer's GUI-facing Façade.

Sane Subset—languages and libraries provide arbitrarily wide design spaces. Only use that subset of possible constructions known to provide the safest results. Stick with what your team knows, and well-defined language constructs.

This book's case studies illustrate decisions based on a Sane Subset, without specifying it. It would fill another book. Software classics, such as the *Effective C++* books by Scott Meyers, or *Code Complete* by Steve McConnell, fill this niche.

Self Test pattern—a source file that, evaluated alone, tests itself. When imported into an application, the tests do nothing.

Side effect—An effect a method has on something other than its return value, such as changes to non-local variables, or outputs.

Simple Design—code with the minimal structure and behavior elements to clearly support only the current set of requirements.

Skin—an alternative GUI, differing by esthetics, usability, or locale.

Static Typing—languages like Java, Eiffel, or C++ bind the type of objects at compile time, and select the target of messages at run time based only on types appearing within an object's inheritance graph.

```
Object &aObject = getObject();
aObject.method(argument);
```

That statement requires `aObject`, at compile time, to refer only to objects created by classes within the same inheritance graph as Object. At runtime, calls to `.method()` dispatch to the correct type of `aObject`, but unrelated objects with a `.method()` could not have compiled.

Sustainable Pace—sleep, snacks, and occasional frantic bouts of Dance Dance Revolution Max2™ are excellent design techniques.

System Metaphor—a common theme for project-specific jargon, to facilitate communication. Our beloved "Uncle" Bob Martin put it best:

> I once worked on a multi-user system where a stream of characters had to be sent down a serial link. We used a double-buffered process such that as one buffer was being filled from the character source, the other was being sent down the serial link. When the sending buffer emptied, the two buffers swapped places.
>
> I tried to explain this to a colleague. At first he had trouble visualizing the system. Then, suddenly, he exclaimed: "It's like two dump trucks! While one is being loaded from a big pile of dirt, the other is dumping that dirt into a pit. When the dumping truck is empty it drives back to the pile. When the loading truck is full, it drives to the pit.
>
> This metaphor made sense to him, but it also allowed him to play games with the concept. He started questioning the number of trucks, or whether or not there could be more than one pile or pit. He started considering the effects of multiple stops for the trucks, and priorities on the trucks, etc. The concept of a truck dispatcher was discussed.
>
> Once a metaphor is in place, it provides the system of names and the essential relationships that the developers can use to discuss the solution.

Temporary Interactive Test—write a test that Regulates Event Queue, turn its spigot on, and instrument the GUI Toolkit's callbacks to write trace statements. If you have a debugger, put breakpoints in the callbacks, too. Run the tests, view a window, interact with it, and use the trace statements and debugging to replace the test with a permanent one that does not require interaction. (If a test fails inexplicably, switch back to interactive mode to see if the behavior is correct but the test is wrong.)

Testage—environment of testing.

Test Case—a single method that tests, typically testing one little thing, and typically a member of a Test Suite.

Test Code—code living on the "test side" of the project; inside Test Cases, Suites, Fixtures and Resources. Distinct from Production Code, which compiles into a deliverable application.

Test Collector—reflection (or Regular Expressions applied to source code) to collect all test cases automatically, without the need to write extra statements that combine them into suites. Page 184 shows this technique for C++, without reflection or Regular Expressions.

Test-First Programming—using automated tests to drive all of specifying, designing, implementing, and delivering a software project. This requires every large-scale design requirement and every small-scale code ability to start with a test and finish with code passing that test and all others.

Test First—when tests lead development. Used as a transitive verb: "I will test-first a new button onto that window."

Test Fixture—behavior reused by test cases. Test Resources are data processed by Test Fixtures. For legacy code, test fixtures may form a Façade Pattern on your application, giving it the interface that tests want.

Some Agile engineers use definitions of "fixture" that overlap mine. My definition is wide, but distinct, and necessary to discuss the aggressive Test Code that GUIs require.

Test Resource—data records that a test fixture processes. Specifically, batches of sample inputs and outputs, all of the same format. Their Test Case traverses their list, and passes each one into a test fixture. This activates the target Production Code, collects its response, and asserts that it matches the resource's sample output.

Some Agile engineers use "Test Resource" to mean data records that test cases share.

Test Rig—the infrastructure supporting assertions—the Test Runner, Test Suites, Test Cases, test fixtures and Test Resources.

Test Runner—the Test Rig "engine" that calls every Test Case in every Test Suite, and reports their results to a console or a GUI.

Transclude—transitive include. Said of an authoring environment that pulls in something from another, remote authoring environment. For example, most Wikis can transclude graphics from other sites. The *Broadband Feedback* Principle leads to these practices.

Turtle Graphics—graphics command sequences that order a hypothetical invisible turtle to move, turn, and draw with a pen. The turtle maintains a current location and orientation, so subsequent commands compound the effects of previous ones. See:

```
http://flea.sourceforge.net/
```

Unit Test—a test whose failure implicates only one unit. This technique forces other units to replace with Mock Objects. Contrast with Developer Test.

User Story—the name of a feature, written on an index card. Location of the card representsthe feature's status within a team's workflow. At implementation time, the card should now be the unimplemented feature with the highest business value. Just before implementation is the most cost-effective time to fully specify the card by writing a Customer Test. The book *User Stories Applied: For Agile Software Development* by Mike Cohn illustrates the universe revolving around these lowly index cards.

Whole Team—traditional teams reward senior engineers with private offices. Development is hard, and should instead use a "continuous meeting", to help the various roles—Customer Team, developers, etc.—learn from each other, in real-time.

Bibliography

For further research: `http://www.google.com/`

Foundational Citations:

- *Refactoring: Improving the Design of Existing Code* by Martin Fowler
- *Extreme Programming eXplained: Embrace Change* by Kent Beck
- *Test Driven Development: By Example* by Beck

Contextual Citations:

- *Accelerated C++: Practical Programming by Example* by Andrew Koenig & Barbara E. Moo
- *Agile Development: Principles Practices and Patterns* by Robert C. Martin
- *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process* by Scott W. Ambler
- *The Algorithmic Beauty of Plants* by Przemyslaw Prusinkiewicz & Aristid Lindenmayer
- *AntiPatterns: refactoring software, architectures, and projects in crisis* by Brown, Malveau, McCormick & Mowbray
- *C++ Coding Standards: Rules, Guidelines, and Best Practices* by Herb Sutter & Andrei Alexandrescu
- *The C++ Programming Language 3rd Edition* by Bjarne Stroustrup
- *C++ GUI Programming with Qt 3* by Jasmin Blanchette & Mark Summerfield
- *Code Complete 2nd Edition* by Steve McConnell
- *Developing International Software 2nd Edition* by Dr. International
- *Design Patterns: elements of reusable object-oriented software* by Gamma, Johnson, Helm, & Vlissides
- *Domain Driven Design: Tackling Complexity in the Heart of Software* by Eric Evans
- *Effective C++ 2nd Edition* by Scott Meyers
- *Exceptional C++* by Herb Sutter
- *Extreme Programming Examined* edited by Giancarlo Succi
- *Extreme Programming for Web Projects* by Doug Wallace, Isobel Raggett, and Joel Aufgang
- *GUI Bloopers* by Jeff Johnson
- *How to Break Software: A Practical Guide to Testing* by James A. Whittaker
- *Internationalization with Visual Basic* by Michael Kaplan

- *JUnit Recipes*: *Practical Methods for Programmer Testing* by J.B. Rainsberger
- *Large Scale C++ Software Design* by John Lakos
- *Lean Software Development*: *An Agile Toolkit for Software Development Managers* by Mary Poppendieck and Tom Poppendieck
- *Modern C++ Design*: *Generic Programming and Design Patterns Applied* by Andrei Alexandrescu
- *More Effective C++* by Scott Meyers
- *More Exceptional C++* by Herb Sutter
- *Notes on the Synthesis of Form* by Christopher Alexander
- *The Pragmatic Programmer*: *From Journeyman to Master* by Andy Hunt & Dave Thomas
- *Programming Ruby*: *The Pragmatic Programmer's Guide* by David Thomas & Andrew Hunt
- *Rapid Development*: *Taming Wild Software Schedules* by Steve McConnell
- *Refactoring to Patterns* by Joshua Kerievsky
- *Scientific and Engineering C++*: *An Introduction with Advanced Techniques and Examples* by John J. Barton, Lee R. Nackman
- *Smalltalk Best Practice Patterns* by Kent Beck
- *Test-First Programming*: *A Practical Guide*, by Dave Astels
- *Testing Extreme Programming*, by Lisa Crispin & Tip House
- *User Stories Applied*: *For Agile Software Development* by Mike Cohn
- *Web Pages that Suck* by Vincent Flanders
- *The Wiki Way*: *Collaboration and Sharing on the Internet* by Ward Cunningham & Bo Leuf
- *Working Effectively with Legacy Code* by Mike Feathers

Back Cover:

Test-Driven Development defends changes from bugs, freeing Agile projects to rapidly and frequently boost their users' productivity and profitability.

Graphical User Interfaces notoriously resist test-first coding. Competitive vendors invest their GUI Toolkits with complex features that resist and complicate Agile development.

Tested code is easy to simplify. Simple code is easy to test. When GUIs lack tests, those extremes can't reinforce each other. Complex and risky code, closest to your users, creates bugs.

This book shows how to write simple tests that force any GUI to improve its appearance, its interactions, and its design. All software, including common and advanced user interfaces, may now join the revolution with Agility's most powerful techniques.

Code samples explore:

- CGI
- COM
- DOM
- GDI
- glTrace
- GNU C++
- GraphViz
- HttpUnit
- ImageMagick
- Java
- JavaScript
- Jemmy
- Konqueror
- Mophun
- MS Windows
- OpenGL
- Perl
- Qt
- Ruby
- Sanskrit
- SVG
- Tk
- Unicode
- Uniscribe
- Visual C++
- WebUnit
- WebInject
- WTL
- XHTML
- XML
- XPath
- XSLT

(Some more than others!)

Case Studies use Test-First Programming to write powerful applications illustrating:

- Animation
- Canvases
- Data Entry
- Embedded Games
- Emulators
- Event-Driven Architectures
- 3D Fractals
- Graphic Editing
- Interactive HTML
- Localization
- Model View Controller
- Progress Bars
- Scripting Layers
- Skins
- Turtle Graphics
- and Web-Enabled Acceptance Test Frameworks that target GUIs.

Complete source code is available online.
http://zeroplayer.com/tfui/TFUI_source.zip

Biography:

Phlip applies his compulsion for art and appreciation of science to deliver advanced and interactive displays of hard data for manufacturing, research, geology, linguistics, and games. Coaching XP techniques increases the odds he can use them himself. He has a Masters degree in Computer Science from the Institute of Strategic Fabrications.