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".

TODO: typeset ﬂ with a font that doesn't fake that ligature with sloppy kerning…

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.

TODO: PDF is squelching the [] dead spot. Please ensure one shows up in print.

Localizing to संस्कृत

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.

TODO is there a latent skateboard in here?

TODO kvetch about the Winword grammar checker wants Western Europe but not Southern Asia with a caps…

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.

TODO 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);
    }

};
```

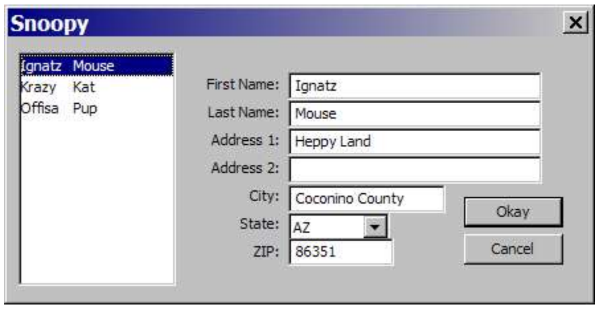
The new suite calls back to its base class's `TestDialog::setUp()` and `Test-Dialog::tearDown()` methods. When they create the dialog member object, its resources select Sanskrit. After the dialog destroys, the suite restores the locale to your desktop's default.

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");
}
```



Ignatz Mouse
Krazy Kat
Offisa Pup

First Name: Ignatz
Last Name: Mouse
Address 1: Heppy Land
Address 2:
City: Coconino County
State: AZ
ZIP: 86351

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.

TODO escalate Resource File

Sanskrit shares a very popular script called देवनागरी ("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 संस्कृत 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 स्पाइडरमैन, 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");
    }
```

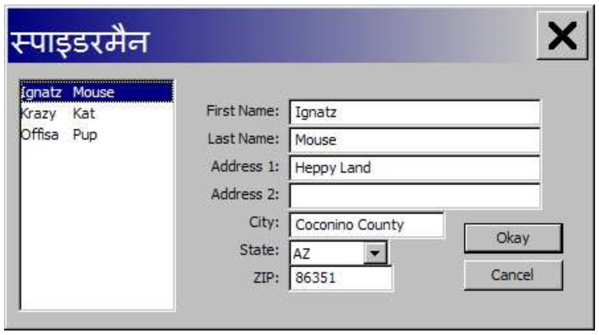
Note that big string does not use the `TEXT()` macro. The width of its characters is not optional.

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 "स्पाइडरमैन"

    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.



स्पाइडरमैन

Ignatz	Mouse
Krazy	Kat
Offisa	Pup

First Name: Ignatz
Last Name: Mouse
Address 1: Heppy Land
Address 2:
City: Coconino County
State: AZ
ZIP: 86351

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:

- स्प **sp**a
- ◌ा *raises the* a *toward* **i**
- इ **ee**
- ड **d**
- र **ar**
- म **m**
- ◌ै *adds* **ai**
- न **n**

Note the ligature between स *sa* and प *pa*, forming स्प *spa*. That ligature suppressed the default *a* sound in स *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 reminds us not to integrate aggressive experiments. Erase this change and start again with a real glossary, in modern Sanskrit:

TODO system to marginalize the “see page” callouts?

TODO: fix the size of the bullet points

उपक्रमः	*upakrama H*	project
नाम	*naama*	name
कुलनाम	*pulanaama*	surname
वीथि १, २	*viithi*	road 1, 2
ग्रामम्	*graamam*	village
ढेशे	*deshe*	land
संहिना	*sa.nhitaa*	code
त्रा	*traa*	save
विराम	*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`	व
`\x093F`	ि

The symbol ि *i* modifies the glyph to its right, but its `\x093F` goes *after* the `\x0935` व *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 क़ॄ. Either all these letters appear very small, or you must set your font to very large.

TODO: Please typeset all the inline Devanāgarī at 14 point (or 2 points bigger than the Latin), if possible without screwing up the paragraph line spacings! The tables use 16 point Devanāgarī.

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.

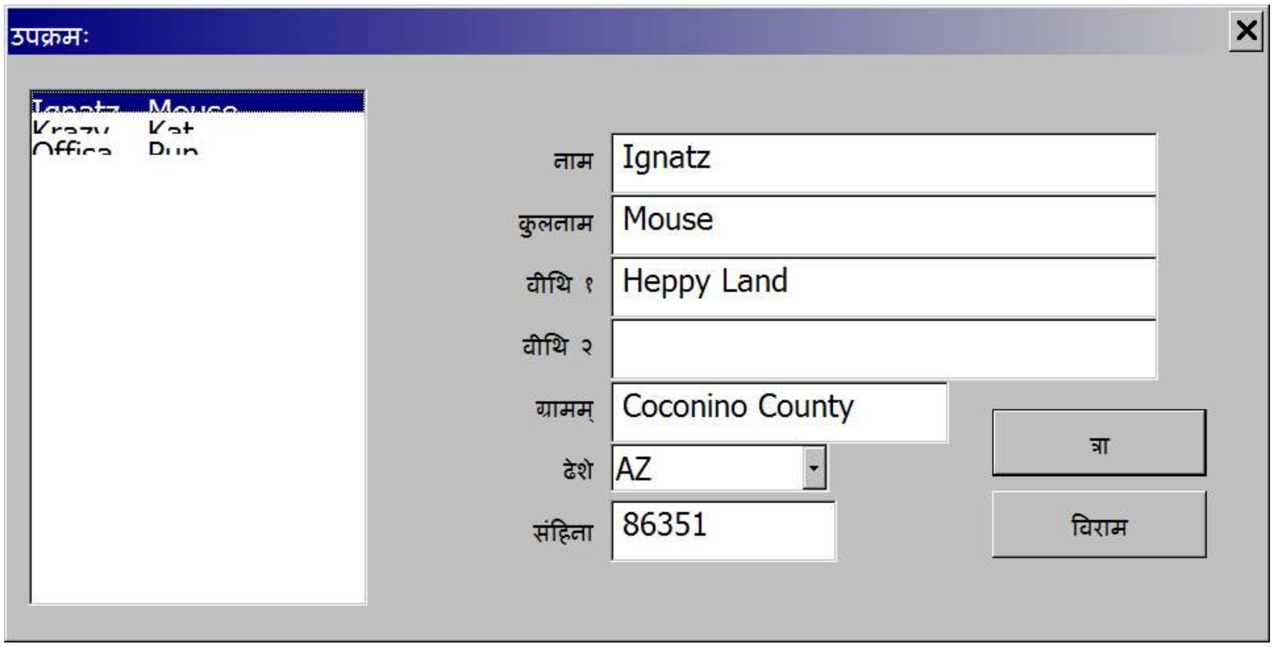
```
TEST_(TestSanskrit, caption)
{

    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:



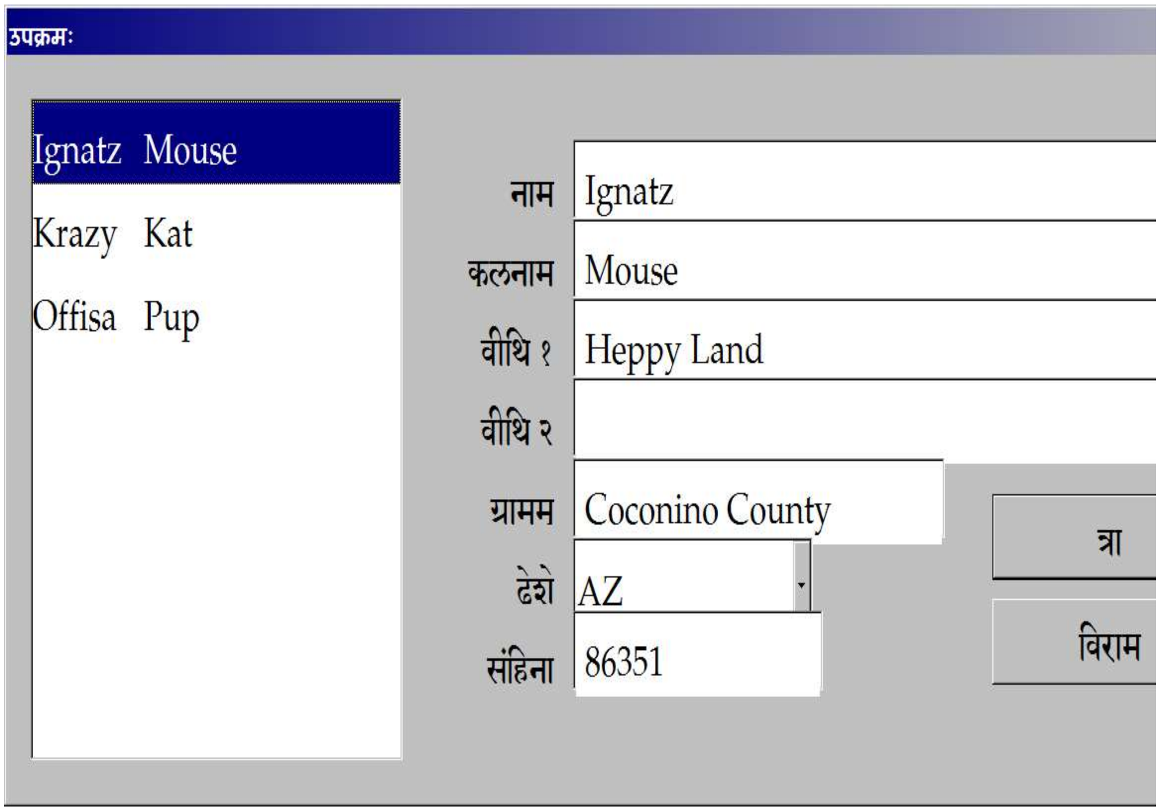
नाम: Ignatz
कुलनाम: Mouse
वीथि १: Heppy Land
वीथि २:
ग्रामम्: Coconino County
ढेशे: AZ
संहिना: 86351

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):


उपक्रमः

Ignatz Mouse
Krazy Kat
Offisa Pup

नाम	Ignatz
कलनाम	Mouse
वीथि १	Heppy Land
वीथि २	
ग्रामम	Coconino County
ढेशे	AZ
संहिना	86351

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 “ि व”, 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.

TODO: PDF is squelching the [] dead spot. Please ensure one shows up in print.

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.)

TODO restore the oldFont?

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), concrete tests for each skin will inherit and specializes a common Abstract Test.

TODO "from" for back-citations, "on" generally for forward citations

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":

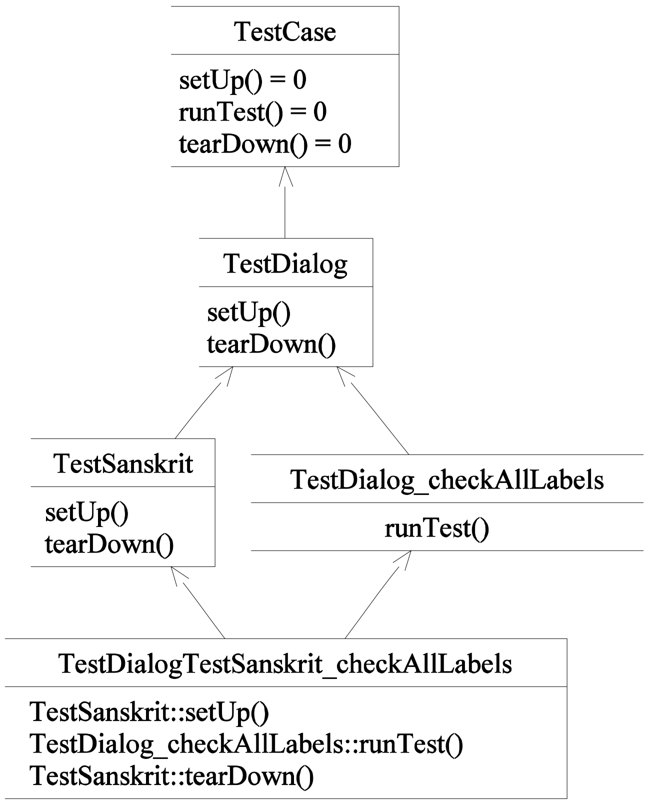
```
#define TEST_(suite, target)              \
    struct suite##target:  virtual suite \
    { void runCase(); }                  \
    a##suite##target;                    \
    void suite##target::runCase()

#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.

TODO your test rig should also provide abstract test by some mechanism

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.

TODO query visual appeanceS ??

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 गुरुन्.