Michael Hunter's

You Are Not Done Yet

checklist

You Are Not Done Yet

Pick something. Anything. A feature in your favorite software application, your favorite toy, your favorite piece of furniture. Now start brainstorming things you could do to test it. Think of as many different things to do to that object as you can. Come back and continue reading when you're done.

What's that? You're back already? There are test cases you haven't thought of, I guarantee it. How do I know? Because for even the tiniest bit of something – the Find dialog box in your web browser, say, there are billions of possible test cases. Some of them are likely to find interesting issues and some of them aren't. Some of them we execute because we want to confirm that certain functionality works correctly. These latter cases are the basis of my You Are Not Done Yet list.

This list is large and can be overwhelming at first. Fear not. You have probably already covered many of these cases. Others won't be applicable to your situation. Some may be applicable yet you will decide to pass on them for some reason or other. Verifying you have executed each of these test cases is not the point of the list. The point is to get you thinking about all of the testing you have and have not done and point out areas you meant to cover which you haven't yet.

So don't quail at the thought of all this testing you haven't done yet. Instead, customize this list to your context. Scratch off items which do not apply. Use the list as a launch point for finding items not on it which do apply. Use it to organize your testing before you start. Use it as a last-minute checklist before you finish. How you use it is not nearly as important as that you use it in the first place.

My original You Are Not Done Yet list was based on similar lists I used within Microsoft. This version has been expanded beyond anything I have seen there, plus I have added explanations for many items.

Input Methods

You are not done testing yet unless...you have tested the following input methods:

- Keyboard. Duh, right? But it's important to remember that testing keyboard input doesn't just mean verifying you can type into text boxes. Scour your application for every different control that accepts text - not just as a value, but also shortcut key sequences and navigation. (Yes, there's some overlap here with Dialog Box Navigation and Accessibility.) If your application uses any custom controls, pay them especial attention as they are likely to use custom keystroke processing. Make those mouse-hating keyboard wizards happy!
- Mouse. Duh again, but again it's so obvious that it's easy to miss. And again, pay especial attention custom controls as they are likely to do custom mouse handling.
- Pen input. Depending on your target platform(s), this could mean pen input direct to your application, filtered through the operating system (e.g., the Tablet Input Panel on Microsoft Windows), and/or filtered through third-party input panels. Each input source has its own quirks that just might collide with your application's own quirks.
- Speech input. Depending on your target platform(s), this could mean speech input direct to your application, filtered through the operating system, and/or filtered through third-party speech processors.
- Foreign language input. On Microsoft Windows this usually means an Input Method Editor (IME), either the one that comes with the operating system or one provided by a third party. These can be troublesome even for applications that do not do any custom keystroke processing. For example, a Japanese-language input processor likely traps all keystrokes, combines multiple keystrokes into a single Japanese character, and then sends that single character on to the application. Shortcut key sequences should bypass this extra layer of processing, but oftentimes they don't. (Note: turning off the IME is one solution to this quandary, but it is almost never the right answer!)
- Assistive input devices such as puff tubes. The operating system generally abstracts these into a standard keyboard or mouse, but they may introduce unusual conditions your application needs to handle, such as extra-long waits between keystrokes.
- Random other input sources. For example, I have seen games where you control the action by placing one or more sensors on your finger(s) and then thinking what you want the program to do. Some of these devices simply show up as a joystick or mouse. What happens if someone tries to use such a device in your application?
- Multiple keyboards and/or mice. Microsoft Windows supports multiple mice and keyboards simultaneously. You only ever get a single insertion point and mouse pointer, so you don't have to figure out how to handle multiple input streams. You may, however, need to deal with large jumps in e.g., mouse coordinates. Oh the testing fun!

Files

You are not done testing unless...you have looked at each and every file that makes up your application, for they are chock full of information which is often ignored. And we all know what happens when

things are ignored - bugs appear! I remember one bug bash where a developer chalked up over fifty bugs simply by going through this list!

- Verify the version number of each file is correct.
- Verify the assembly version number of each managed assembly is correct. Generally the assembly version number and the file version number should match. They are specified via different mechanisms, however, and must explicitly be kept in sync.
- Verify the copyright information for each file is correct.
- Verify each file is digitally signed - or not, as appropriate. Verify its digital signature is correct.
- Verify each file is installed to the correct location. (Also see the Setup YANDY.)
- Verify you know the dependencies of each file. Verify each dependency is either installed by your setup or guaranteed to be on the machine.
- Check what happens when each file - and each of its dependencies - is missing.
- Check each file for recognizable words and phrases. Determine whether each word or phrase you find is something you are comfortable with your customers seeing.

Filenames

You are not done testing yet unless...you have tested the following test cases for filenames:

- Single character filenames
- Short filenames
- Long filenames
- Extra-long filenames
- Filenames using text test cases
- Filenames containing reserved words
- Just the filename (file.ext)
- The complete path to the file (c:\My\Directory\Structure\file.ext)
- A relative path into a subfolder (Sub\Folder\file.ext)
- A relative path into the current folder (.\file.ext)
- A relative path into a parent folder (..\Parent\file.ext)
- A deeply nested path (Some\Very\Very\Very\Very\Very\Deeply\Nested\File\That\You\Will\Never\Find\Again\file.ext)
- UNC network paths (\\server\share\Parent\file.ext)
- Mapped drive network paths (Z:\Parent\file.ext)

Filenames are interesting creatures and a common source of bugs. Microsoft Windows applications that don't guard against reserved words set themselves up for a Denial Of Service attack. Applications on any operating system that allow any old file to be opened/saved/modified leave a gaping hole onto "secured" files. Some users stuff every document they've ever created into their user folder. Other users create a unique folder for each document. Certain characters are allowed in filenames that aren't

allowed elsewhere, and vice versa. Spending some focused time in this area will be well worth your while.

Filename Invalid Characters and Error Cases

You are not done testing yet unless...you have checked for invalid characters in filenames, and for reserved filenames. Operating systems tend to get grumpy if you try to use wildcards (e.g., '*') in filenames. They may also treat certain filenames specially. For example, Microsoft Windows provides a single API for creating/opening files, communication ports, and various other cross-process communication mechanisms. Well-known communication ports (e.g., COM1) are addressed by "filename" just as though they were a file - kinda handy, but it means that you can't use "COM1" for a physical file on disk.

Testing for this is easy: brainstorm a list of interesting test cases, then slap each one into each of your application's dialog boxes, command line arguments, and APIs that take a filename. Illegal characters will probably throw an error, but trying to open a reserved filename is likely to hang your app.

See the MSDN topic "Naming a file" [http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/fs/naming_a_file.asp] for the full skinny on reserved characters and filenames on Microsoft operating systems.

File Operations

You are not done testing unless...you have thoroughly tested your application's Open, Save, and Save As functionality. I don't know about you, but I get grumpy when my work disappears into thin air! For many applications, if data cannot be saved and later regurgitated with full fidelity, the application may as well not exist. Thus it is important to verify the correct thing happens under the following conditions:

- Open each supported file type and version and Save As each supported file type and version. Especially important is to open from and save as the previous version of your native format. Customers tend to get grumpy if upgrading to a new version means they can no longer open old documents! And they tend to not upgrade if they do not have a simple way to share documents created in the new version of your application with those poor souls still languishing on the old version.
- Open each supported file type and version and Save. If the file type and version can be selected during a Save operation (as opposed to a Save As operation), Save to each supported file type and version. More usually, Save saves to the current version only.
- Roundtrip from each supported version to the current version and back to the previous version. Open the resulting file in that version of your application. Does it open correctly? Are new features correctly converted to something the previous version understands? How are embedded objects of previous versions handled?
- Open files saved in the current version of your application in previous versions of your application. If the document opens, how are features added in the new version handled? If the document does not open, is the resulting error message clear and understandable?

- Open from and Save and Save As to different file systems (e.g., FAT and NTFS) and protocols (e.g., local disk, UNC network share, http://). The operating system generally hides any differences between types of file systems; your application probably has different code paths for different protocols however.
- Open, Save, and Save As via the following mechanisms (as appropriate):
    - Menu item
    - Toolbar item
    - Hot key (e.g., Control+S for Save)
    - Most Recently Used list
    - Microsoft SharePoint document library
    - Context menu(s)
    - The application's Most Recently Used list
    - The operating system's Most Recently Used list
    - Drag-and-drop from the file system explorer
    - Drag-and-drop from your desktop
    - Drag-and-drop from another application
    - Command line
    - Double-click a shortcut on your desktop
    - Double-click a shortcut in an email or other document
    - Embedded object
- Open from and Save and Save As to the following locations:
    - Writable files
    - Read-only files
    - Files to which you do not have access (e.g., files whose security is set such that you cannot access them)
    - Writable folders
    - Read-only folders
    - Folders to which you do not have access
    - Floppy drive
    - Hard drive
    - Removable drive
    - USB drive
    - CD-ROM
    - CD-RW
    - DVD-ROM
    - DVD-RW
- Open from and Save and Save As to various types and speeds of network connections. Dial-up and even broadband has different characteristics than that blazing fast one hundred gigabyte network your office provides!
- Open files created on (and Save and Save As to as appropriate):
    - A different operating system

    - An OS using a different system locale
    - An OS using a different user locale
    - A different language version of your application
- Open from and Save and Save and Save As to filenames containing
    - The Text Entry Field YANDY list, as appropriate
    - The Filenames YANDY list, as appropriate
    - The Invalid Filenames YANDY list
    - Spaces
- Cause the following to occur during Open, Save, and Save As operations:
    - Drop all network connections
    - Fail over to a different network connection
    - Reboot the application
    - Reboot the machine
    - Sleep the machine
    - Hibernate the machine
- Put AutoSave through its paces. What happens when you AutoSave every zero minutes? Every minute? With a very big document? If the AutoSave timer is per document, what happens when multiple AutoSaves kick off simultaneously, or while another AutoSave is in progress? Does file recovery from AutoSave work as you expect? What happens if the application crashes during an AutoSave? During recovery of an AutoSaved document?
- Save and Save as in the following conditions:
    - No documents are dirty
    - One document is dirty
    - Multiple documents are dirty and the user chooses to save all of them
    - Multiple documents are dirty and the user chooses to save none of them
    - Multiple documents are dirty and the user chooses to save only some of them

Network Connectivity

You are not done testing yet unless...you have verified how your application handles various network configurations and events. In times past you could more or less count on stability in the network - if the computer was connected to a network when your application started, it would almost certainly remain connected to that network while your application was open. Sure, some doofus might cut the main feed with a backhoe or yank the wrong cable in the router closet. The chances of something catastrophic happening were low enough, however, that bugs of the form "Disconnect your computer from the network while the application is opening a twenty megabyte file from a network share" tended to be Won't Fix'd with dispatch under the premise that "No user is going to do that".

Oh how times have changed! Users these days are more likely than not to be connected to a wireless network which drops out on a regular basis. Users who start out connected to a wired connection may undock their computer and thus disconnect from that network at any time. Net-over-cell phone is becoming ever more prevalent. Network-related "We'll fix that if we have time, maybe" issues are now

problems which directly affect your customers on a regular basis. And so it is important to check the following:

- Connecting over a network which supports only IPv4
- Connecting over a network which supports only IPv6
- Connecting over a network which supports both IPv4 and IPv6
- Connecting over an 802.11a wireless network
- Connecting over an 802.11b wireless network
- Connecting over an 802.11g wireless network
- Connecting over an 802.11n wireless network
- Connecting over a GPRS (cell phone) network
- Connecting from a multi-homed machine (i.e., one which is connected to multiple networks)
- Connecting via a 28.8 modem
- Connecting via a 56k modem
- Connecting over a network other than the one inside your corporate firewall
- Connecting over a network which requires user authentication on first access
- Connecting over a network which requires user authentication on every access
- Passing through a software firewall
- Passing through a hardware firewall
- Passing through Network Address Translation
- Losing its connection to the network
- Losing its authority to connect to the network
- Joined to a workgroup
- joined to a domain
- Accessing documents from a network location which requires user authentication
- Performing a Print Preview to a network printer which is disconnected or otherwise unavailable

Alerts

You are not done testing yet unless…you have searched out every alert, warning, and error message and dialog box your application can display and checked the following:

Content

- Verify that you understand every condition that can cause the alert to display, and that you have test cases for each condition (or have explicitly decided to *not* test specific conditions).
- Verify that the alert is in fact needed. For example, if the user can easily undo the action, asking them whether they really want to do it is not necessary.
- Verify that the alert first identifies the problem and then presents the solution. Basically, treat your customers like smart, knowledgeable people and help them understand what the problem is and what they can do about it.
- Verify that the alert text does not use an accusatory tone but rather is polite and helpful. I remember one application who scolded me with "You did not close me correctly" as it auto-recovered its database after crashing. Umm, no, *you* did not close yourself correctly; I had

nothing to do with it! A later release of the application changed the message to "This application was not closed correctly", which is a little better. As was the case for me, almost certainly your customer did not cause the problem intentionally. If they did do it on purpose, likely they didn't know it would be a problem. Again, let them know what happened, what the application is doing to remedy the situation, and what they can do to prevent it from happening in the future.

- Verify the alert text is correct and appropriate for the situation.
- Verify the alert text is consistent in its wording and style, both to itself as well as to each other alert.
- Verify the alert text is as succinct as possible but no more succinct. Hint: If the alert text is longer than three lines, it's probably too long.
- Verify the alert text contains complete sentences which are properly capitalized and punctuated.
- Verify the alert text does not use abbreviations or acronyms. (Discipline-specific acronyms may be OK, if you are confident that all of your users will know what they mean.)
- Verify the alert text uses the product's name, not pronouns such as "we" or "I".

Functionality

- Verify the alert's title bar contains the name of the product (e.g., "Acme Word Processor").
- Verify each button works correctly. I have seen innumerable "Cancel" buttons that were really an "OK" button in disguise!
- Verify each button has a unique access key.
- Verify the buttons are centered below the message text.
- Verify any graphics on the alert are appropriate and correctly placed. For Microsoft Windows applications, there are standard icons for Informational, Warning, and Critical alerts, and these icons are typically displayed to the left of the alert text.

Accessibility

You are not done testing yet unless...you have verified your application plays nicely with the accessibility features of your operating system. Accessibility features are vital to customers who are blind, deaf, or use assistive input devices, but they are also extremely useful to many other people as well. For example, comprehensive large font support will be much appreciated by people with failing eyesight and/or high DPI screens.

Some of the following terms and utilities are specific to Microsoft Windows; other operating systems likely have something similar.

- Verify that every control on every dialog and other user interface widget supports at least the following Microsoft Active Accessibility (MSAA) properties:
    - Name - its identifier
    - Role - a description of what the widget does, e.g., is it invokable, does it take a value
    - State - a description of its current status
    - Value - a textual representation of its current value

    - KeyboardShortcut - the key combination that can be used to set focus to that control
    - DefaultAction - a description of what will happen if the user invokes the control; e.g., a checked check box would have a Default Action of "Uncheck", and a button would have a Default Action of "Press"
- Verify that changing the value of each control updates its MSAA State and Value properties.
- Run in high contrast mode, where rather than a full color palette you have only a very few colors. Is your application still functional? Are all status flags and other UI widgets visible? Are your toolbars and other UI still legible? Does any part of your UI not honor this mode?
- Run in large font mode, where the system fonts are all extra large. Verify that your menus, dialogs, and other widgets all respect this mode, and are still legible. Especially be on guard for text that is truncated horizontally or vertically. To really stress your UI, do this on a pseudo-localized build!
- Run with Sound Sentry, which displays a message box, flashes the screen, or otherwise notifies the user anytime an application plays a sound. Verify that any alert or other sound your application may play activates Sound Sentry.
- Run with sticky keys, which enables the user to press key chords in sequence rather than all at once. The operating system will hide much of these details from your application, but if your app ever directly inspects key state it may need to explicitly handle this state.
- Run with mouse keys, which enables the user to control the mouse pointer and buttons via the numeric keypad. Again, the operating system will hide much of these details from your application, but if your app ever directly inspects mouse state it may need to explicitly handle this state.
- Run with no mouse and verify that every last bit of your UI can be accessed and interacted with solely through the keyboard. Any test case you can execute with a mouse should be executable in this mode as well.
- Run with a text reader on and your monitor turned off. Again, you should be able to execute each of your test cases in this state.
- Verify focus events are sent when each control loses and receives focus.
- Verify the tabbing order for each dialog and other tab-navigable UI component is sensible.
- Verify that any actionable color item (e.g., that red squiggly line Microsoft Word displays underneath misspelled words) can have its color customized.
- Verify that any object which flashes does so to the system cursor blink rate.

How completely you support these various accessibility features is of course a business decision your team must make. Drawing programs and other applications which incorporate graphics, for example, may decide to require a mouse for the drawing bits. As is also the case with testability, however, accessibility-specific features are often useful in other scenarios as well. (The ability to use the keyboard to nudge objects in drawing programs tends to be popular with customers of all abilities, for example.)

Text Accessibility

You are not done testing yet unless...you have checked that all text is actually text and not a bitmap or video. Text rendered as a graphic is problematic for two reasons. First, accessibility clients such as

screen readers can't see into bitmaps, videos, and animations, so any text embedded in such a graphic is invisible to anyone using accessibility features. Second, graphics with embedded text vastly complicate the localization process. Translating text simply requires modifying the application's resource files, but translating bitmaps and videos requires recompositing them.

If you must place text in bitmaps, you can mitigate your localization pain by creating the bitmaps dynamically at runtime using resourced text strings. Videos and animations may be runtime creatable as well depending on the toolset you use.

As for accessibility, ensure that the relevant information is available some other way: in the supporting text, by closed captioning, ALT tags in HTML, and so on.

Menus and Command Bars

You are not done testing yet unless...you have put your menus and command bars through their paces. There used to be a distinct difference between menus and command bars: menus could have submenus and were always text (perhaps with an optional icon) while command bars were never nested and were only graphics. Nowadays, however, menus and toolbars are more-or-less the same animal and can be mixed-and-matched, so that the only real difference is that command bars are typically always visible whereas menus are transient.

- Verify all commands work from menus and from command bars
- Verify each keyboard shortcut works correctly
- Verify built-in commands work correctly from a custom menu
- Verify built-in commands work correctly from a custom command bar
- Verify custom commands work correctly from a custom menu
- Verify custom commands work correctly from a custom command bar
- Verify custom commands work correctly from a built-in menu
- Verify custom commands work correctly from a built-in command bar
- Verify custom menus and command bars persist correctly
- Verify customizations to built-in menus and command bars persist correctly
- Verify commands hide/disable and show/enable as and only when appropriate
- Verify command captions are correct and consistent with similar terms used elsewhere
- Verify menu and command bar item context menus work correctly (although menus on menus is going a bit too far I think)
- Verify status bar text is correct
- Verify status bar text is not truncated

Dialog Box Behavior

You are not done testing yet unless…you have checked the following points for each and every dialog box in your application:

- Verify that each command (e.g., menu item, shortcut key) that is supposed to launch the dialog box does in fact launch it.

- Verify that its title is correct.
- Verify that all terms used by the dialog are consistent with those used by the rest of the application.
- Verify that accepting the dialog box updates application state correctly.
- Verify that canceling the dialog box causes no change to application state.
- Verify that the dialog is sticky - displays in the position from which it was last dismissed. Or that it always displays in the same location, if it is not supposed to be sticky.
- Verify that the dialog's contents are initialized from the current state of the application. Or that it always starts with default values, if it is not supposed to initialize from application state.
- Verify that invoking help (e.g., pressing F1) links to the correct help topic. Note that you may need to check this for each individual control as some dialog boxes have control-specific context-sensitive help.

Dialog Box Interactivity

You are not done testing yet unless…you have checked the following points for each and every dialog box in your application:

- Verify that the correct system controls display on the title bar (e.g., some dialog boxes can be maximized while others cannot) and work correctly.
- Verify that the default edit-focused control and default button-focused control are correct.
- Verify that the dialog can be canceled by
    - Pressing the Escape key (regardless of which control has focus)
    - Pressing the system close button (the 'x' button on Microsoft Windows dialog boxes)
    - Pressing the cancel button on the dialog
- Verify that the dialog can be closed and its contents accepted by
    - Pressing the Enter button (regardless of which control has focus, although multi-line text boxes may need an exemption)
    - Pressing the accept or OK button on the dialog
- Verify that the keyboard navigation order is correct. (Microsoft Windows apps often refer to this as "tab order" as the convention on that operating system is that pressing Tab moves you through the dialog box.)
- Verify that every control has a shortcut letter, that every shortcut works, and that each shortcut is unique within the dialog box.
- Verify that each control's tooltip is correct.
- Verify that any mutually exclusive controls work together correctly.
- Verify all dialog states, such as different sets of controls being visible due to application state or "more details" and "less details" buttons on the dialog box being invoked.
- Verify that all ninchable controls (controls which can be in an indeterminate state; for example, the Bold button would be ninched if the current selection contains some text that is bolded and some text that is not bolded) do in fact ninch as appropriate.

- Verify that editing a ninched value has the correct effect (i.e., applies the new value(s) to all items which should be affected; to revisit the text example, all text should be bolded).
- Verify that each control responds correctly to valid input and invalid input, including appropriate boundary cases. For example, invalid input might cause a message box to be displayed, or highlight the control in some fashion.
- Verify that the dialog displays and functions correctly
    - With different color settings
    - With different font settings
    - In high contrast mode
    - In high DPI mode
- Verify that all images and other media in the dialog box are localized correctly.

Dialog Box Look And Feel

You are not done testing yet unless…you have checked the following points for each and every dialog box in your application:

- Verify that the menu command which launches the dialog ends with an ellipsis (e.g., "Create New Document..."). This is the convention on Microsoft Windows at least; for other operating systems check your style guide.
- Verify that the size of each control and spacing between each pair of controls matches that specified by your style guide.
- Verify that the dialog box is sized correctly relative to its controls.
- Verify that ninchable controls display correctly (per your style guide) when they are ninched. (Generally, they should grey out or otherwise make obvious their ninched state.)
- Verify that any samples in the dialog reflect the actual contents and formatting of the current document. Or reconsider showing samples! Dialogs which affect document formatting often purport to preview the effect its settings will have on the active document. Bringing the actual document (or an appropriate piece, such as the current selection) into the preview greatly enhances the value of the preview. If your preview simply presents some preset, made up content that hopefully looks somewhat like the real document, you might as well not have a preview at all.

Although this fit-and-finish stuff can seem like a waste of time, it matters. Although they likely aren't conscious of it, these details affect people's evaluation of your product's quality just as much as how often it crashes does. In fact, if the first impression a potential customer has is that your application is unpolished, they will tend to view the rest of their experience through that lens as well. So polish that chrome!

Text Entry Fields

You are not done testing yet unless...you have covered the following boundary conditions for every text entry field in your application. (Don't forget about editable combo boxes!)

- Null (if you are testing an API)

- Zero characters
- One character
- Two characters
- Some characters
- Many characters
- One less than the maximum allowed number of characters
- The maximum allowed number of characters
- One more than the maximum allowed number of characters
- Spaces in the text
- Symbols (e.g., colon, underscore) in the text
- Punctuation in the text
- ASCII characters
- High ASCII characters
- German characters
- Japanese characters
- Hebrew characters
- Arabic characters
- Unicode characters from multiple character ranges
- Control characters

Text handling can be fraught with errors. If your application is one hundred percent Unicode, count yourself lucky. Even then, however, you may have to import to or export from non-Unicode encodings. If your application handles ASCII text then you get the fun of testing across multiple code pages (try switching code pages while entering text and see what happens!). And if your application uses double-byte or multi-byte encodings, you may find yourself thinking about switching careers!

Undo and Redo

You are not done testing yet unless...you have tested undo and redo. If your application doesn't support undo then you're off the hook. Otherwise, be sure you've done the following:

- Considered whether each user action should be undoable.
- Considered whether each user actions should be redoable.
- Tested one level of undo
- Tested multiple levels of undo
- Tested one level of redo
- Tested multiple levels of redo
- Redo more times than you've undone. In some applications redo is more a "do again".
- Tested intermixed undos and redos
- Verified that each undoable and redoable command is listed correctly in the undo and redo UI
- Tested undo and redo across document saves (some applications toss their undo and redo stacks when you save)

- Tested undo and redo across document close+reopen
- Tested undo and redo across builds, if your application builds code or uses built code (such as allowing the user to reference custom control libraries). The issue here is that the contents of that built code might change - how do you redo an addition of a custom control that no longer exists in the library?

Simple undo/redo testing is easily done manually and will usually find bugs. These bugs are typically simple programmer errors which are easy to fix. The really interesting bugs are usually found by intermixing undos and redos. This can certainly be done manually, but this is one case where automated test monkeys can add value.

You can decide to have one person test undo and redo across your entire application; I find it works best to have each person test undo and redo for their areas.

Printing

You are not done testing yet unless...you have checked how your application handles printing. If you remember back to the Bad Old Days before your operating system abstracted away (most of) the differences between printers, so that each application had to know intimate details about every printer it might be used with, you surely know how good you have it. That just gives you more time to worry about the following issues.

- Verify changing orientation works properly. Try doing this for a brand new document and for an in-progress document. Also try doing this by launching your app's equivalent of a page setup dialog box both directly (e.g., from a menu item) and from within the print dialog.
- Verify printing to a local printer works properly.
- Verify printing to a network printer works properly.
- Verify printing to a file works properly. Every operating system I know of allows you to create a print file for any printer you have installed.
- Verify printing to a PCL printer works properly. PCL started out as the control language for Hewlett-Packard printers but has since become somewhat of a standard.
- Verify printing to a PostScript printer works properly. This printer control language was created by Adobe and has also become somewhat of a standard. PostScript is semi-human readable, so you can do some printer testing by inspecting the output file and thus avoid killing any trees.
- Verify printing to a PDF file works properly. There are a number of free and low-cost PDF creators available; also consider purchasing a copy of Adobe Acrobat in order to test the "official" way to create PDFs.
- Verify canceling an in-progress print job works properly. My current printer pretends to let me cancel, but then pages come spitting out anyway. Frustrating!
- Verify setting each print option your application supports has the proper effect; number of copies, collation, and page numbering, for example.
- Verify setting printer-specific options works properly. These settings should be orthogonal to your application's print settings, but you never know.

Although it may seem that some of this testing should be taken care of by your operating system's testers, I find that developers seem to always have some little customization they make to these dialogs, and so even though it appears to be a standard dialog something is different. These little tweaks can turn out to be bug farms, I think in part precisely because the dev is thinking that it's such a small thing nothing can go wrong.

Even when I really do have a standard dialog box I like give it a once-over , just as a sanity check. The same applies to any printer-specific options. Everything *should* work correctly, but I'm a lot happier when I *know* it does!

In the general case, it's a risk assessment you and your feature team have to make. Bugs *could* be anywhere; where do you think they most likely are? Hit those areas first, and then cover the next most likely, and then the next most likely, and so on. Mix in some exploratory testing too, since bugs have a penchant for cropping up in places you wouldn't think to look for them!

Special Modes And States

You are not done testing yet unless...you have tested your application in the following special modes and states. Ideally you would run each of your tests in each of these special cases, but I haven't yet met anyone who has that much time. More typical is to pick one case each day as the context in which to run your tests that day.

- Different zoom levels, as appropriate. One application I know of had a huge number of automated tests, and everything seemed hunky dory. Then someone thought to try changing the zoom level and found all sorts of issues. Oops.
- Safe Mode. Microsoft Windows has a special mode where just the essentials are loaded - the most basic display driver, a bare-bones network stack, and no start-on-boot services or applications. How does your app handle being run under these conditions?
- Sharing documents between multiple users and/or multiple machines, simultaneously and sequentially. This is especially important for programs that access a database (what happens when someone else makes a change to the record you are editing?), but if you can open documents off a network share, or you can open documents from a shared location on the local machine, someone else can do so as well - potentially the very same document you are editing.
- No file open, dirty file open, dirty-but-auto-saved file open, saved file open.
- Full screen and other view modes.
- Different application window sizes (document window sizes too, if your app has a multi-document interface); especially: default launch size, minimized, maximized, not-maximized-but-sized-to-fill-the-screen, and sized very small.
- Invoke standby, hibernation, and other power-saving modes whilst an operation is in progress. I discovered the hard way that when one application has a modal dialog up it blocks my OS from sleeping. I was *not* a happy camper!
- Resume your computer out of various sleep modes. Do in-progress operations continue where they stopped? Or do they restart? Or do they hang?

- Modified system settings. Set your mouse to move faster or slower. Change your keystroke repeat duration. Mess with your system colors. Does your application pick up the new values when it starts? Does it pick up values that change while it's running?
- Object Linking and Embedding (OLE). If your app supports OLE, you're in for a world of painfun! Does embedding other OLE objects in your app's documents work correctly? What about embedding your app's documents in other OLE-enabled applications? Do embedded applications activate and deactivate correctly? Do linked OLE documents update when the source of the link is modified? How does your app handle the linked document's application not being available?
- Multiple selection. What happens if you apply text formatting when you have three different text ranges selected? Or you paste when several different items are selected? What should happen?

The last two special states are not contexts in which to execute your test cases but rather additional tests to run at the end of each of your test cases:

- Send To. Many applications today have a handy menu item that lets you send the current document to someone as an email. I've seen nasty bugs where attempting to use this handy menu item crashes the application or worse.
- Cut, copy, and delete. To and from the same document, a different document, competing applications, targets that support a less-rich or more-rich version of the data (e.g., copying from a word processor and pasting into a text editor), targets that don't support any version of the data (what happens if you copy from your file explorer and paste into your application?)...you get the idea.

International Sufficiency

You are not done testing yet unless...you have vetted your application's readiness for use in international locales. Even if you are positive that your application will never be used anywhere other than your own country, these issues are worth at least investigating. Your team may decide not to fix problems you find, but at least y'all will know where they are. And, really, are you one hundred percent certain that someone from upper Elbonia *won't* want to use your product?

- Look for culture-specific images and terminology. For example, the color red is synonymous with danger in many Western cultures, and it symbolizes happiness or good luck in others.
- Look for geo-political sensitivity issues. One common example is maps that cover areas whose ownership or exact boundaries are disputed between countries. If your application contains or displays maps in any fashion, be prepared for all kinds of pain the moment anyone outside your country starts using them!
- Verify that your application correctly handles switching to different system locales, language packs, and code pages, both before your application has started and while it is running.
- Verify that your application correctly handles switching to different regional settings, both before your application has started and while it is running: date and time formats, currency symbols and formats, and sort orders, to name just a few. Some or all of these settings will vary

across locales and language packs; most modern operating systems allow you to customize all of this separately from changing languages or locales as well. (On Microsoft Windows, do this via the Regional Settings control panel applet.) For example, if your application works with currency, see what happens when you change your currency symbol to "abc". One application I worked on did not like this at all!

- Verify that your application correctly handles multi-byte (e.g., Japanese), complex script (e.g., Arabic) and right-to-left (e.g., Hebrew) languages. Can you cursor around this text correctly? What happens if you mix right-to-left and left-to-right text?

- Verify that all controls correctly interact with Input Method Editors (IMEs). This is especially important if you intend to sell into East Asian countries.
- Verify that your application correctly handles different keyboard mappings. As with regional settings, certain locales and language packs will apply special keyboard mappings, but operating systems usually allow you to directly modify your keyboard map as well.
- Verify your application correctly handles ANSI, multi-byte, and Unicode text, extended characters, and non-standard characters on input, display, edit, and output.
- Verify that the correct sorting order is used. Sorting correctly is hard! Just ask anyone who has run into the infamous Turkish "i" sort order bug. If you rely on operating system-provided sort routines then you should be in good shape, but if your application does any custom sorting it probably does it wrong.
- Verify that the system, user, and invariant locales are used as appropriate: use the user locale when displaying data to the user; use the system locale when working with non-Unicode strings, and use the invariant locale when formatting data for storage.
- Verify that any language-dependent features work correctly.
- Verify that your test cases correctly take into account all of these issues. In my experience, testers make all the same mistakes in this area as do developers - and won't you be embarrassed if your developer logs a bug against your test case!

Localization

International sufficiency testing is important for just about any application, but localization testing kinda only matters if you are localizing your application into other languages. The distinction can be hard to remember, but it's really quite simple: international sufficiency testing verifies that your application does not have any locale-specific assumptions (like expecting the decimal separator to be a decimal point), whereas localization testing verifies your application can be localized into different languages. Although similar the two are completely orthogonal.

The simplest way to get started localization testing is with a pseudo-localized (aka pseudoloc) build. A pseudoloc build takes your native language build and pseudo-localizes it by adding interesting stuff to the beginning and end of each localized string (where "interesting stuff" is determined by the languages to which your product will be translated, but might include e.g. double-byte or right-to-left characters). This process can vastly simplify your localization testing:

- It allows every build to be localized via an automated process, which is vastly faster and cheaper than is the case when a human hand localizes.
- It allows people who may not read a foreign language to test localized builds.
- Strings that should be localized but are not are immediately obvious as they don't have extra characters pre- and post-pended.
- Strings that should not be localized but in fact are do have extra characters pre- and post-pended and thus are also immediately obvious.
- Double-byte bugs are more easily found.
- UI problems such as truncated strings and layout issues become highly noticeable.

If you can, treat pseudoloc as your primary language and do most of your testing on pseudoloc'd builds. This lets you combine loc testing and functionality testing into one. Testing on actual localized builds - functionality testing as well as localization test - is still important but should be trivial. If you do find major localization bugs on a localized build, find a way to move that discovery into your pseudoloc testing next time!

Beyond all that, there are a few specific items to keep in mind as you test (hopefully pseudo-)localized builds:

- Verify each control throughout your user interface (don't forget all those dialog boxes!) is aligned correctly and sized correctly. Common bugs here are auto-sizing controls moving out of alignment with each other, and non-auto-sizing controls truncating their contents.
- Verify all data is ordered/sorted correctly.
- Verify tab order is correct. (No, this shouldn't be affected by the localization process. But weirder things have happened.)
- Verify all strings that should have been localized were. A should-have-been-localized-but-was-not string is likely hard-coded.
- Verify no strings that should not have been localized were.
- Verify all accelerator key sequences were localized.
- Verify each accelerator key sequence is unique.
- Verify all hot key combination were localized.
- Verify each hot key combination is unique.

APIs

If your application installs EXEs, DLLs, LIBs, or any other kind of file - which covers every application I've ever encountered - you have APIs to test. Possibly the number of APIs *should* be zero - as in the case of for-the-app's-use-only DLLs, or one - as in the case of an EXE which does not support command line arguments. But - as every tester knows - what should be the case is not always what is the case.

- Verify that all publicly exposed APIs should in fact be public. Reviewing source code is one way to do this. Alternatively, tools exist for every language to help with this - Lutz Roeder's .Net Reflector is *de rigueur* for anyone working in Microsoft .Net, for example. For executables, start by invoking the application with "-<command>", ":<command>", "/<command>",

"\<command>" and "<command>" command line arguments, replacing "<command>" with "?" or "help" or a filename. If one of the help commands works you know a) that the application does in fact process command line arguments, and b) the format which it expects command line arguments to take.

- Verify that no non-public API can cause harm if accessed "illegally". Just because an API isn't public does not mean it can't be called. Managed code languages often allow anyone who cares to reflect into non-public methods and properties, and vtables can be hacked. For the most part anyone resorting to such tricks has no cause for complaint if they shoot themselves in their foot, but do be sure that confidential information cannot be exposed through such trickery. Simply making your decryption key or license-checking code private is not sufficient to keep it from prying eyes, for example.
- Review all internal and external APIs for your areas of ownership. Should they have the visibility they have? Do they make sense? Do their names make clear their use and intent?
- Verify that every public object, method, property, and routine has been reviewed and tested.
- Verify that all optional arguments work correctly when they are and are not specified.
- Verify that all return values and uncaught exceptions are correct and helpful.
- Verify that all objects, methods, properties, and routines which claim to be thread safe in fact are.
- Verify that each API can be used from all supported languages. For example, ActiveX controls should be usable (at a minimum) from C++, VB, VB.Net, and C#.
- Verify that documentation exists for every public object, method, property, and routine, and that said documentation is correct. Ensure that any code samples in the docs compile cleanly and run correctly.

Platform

You are not done testing unless...you have considered which platforms to include in and which to omit from your test matrix. The set of supported platforms tends to vary widely across contexts - a consumer application will likely have a different set of supported platforms than does an enterprise line of business application. Even if you officially support only a few specific platforms, it can be useful to understand what will happen if your application is installed or executed on other platforms. Platforms to consider include:

- All supported versions of Windows; at a minimum: Windows XP SP2, Windows XP SP<latest>, Windows Server 2003 SP<latest>, Windows Vista SP<latest>
- Apple OS X.<latest>
- Your favorite distribution of Linux
- Your favorite flavor of Unix
- 32-bit version of the operating system running on 32-bit hardware
- 32-bit version of the operating system running on 64-bit hardware
- 64-bit version of the operating system running on 64-bit hardware
- The various SKUs of the operating system
- Interoperability between different SKUs of the operating system

- Interoperability between different operating systems (e.g., using a Windows Vista machine to open a file which is stored on a Linux network share)
- All supported browsers and browser versions; at a minimum: Internet Explorer 6, Internet Explorer 7, Opera, FireFox
- With and without anti-virus software installed
- With and without firewall software installed

Also peruse the Windows Logo requirements. Even if you aren't pursuing logo compliance (or your application doesn't run on Windows) these are a useful jumping off point for brainstorming test cases!

CPU Configurations

You are not done testing yet unless...you have tested across multiple CPU configurations. A common pitfall I often see is having every last developer and tester use exactly the same make, model, and configuration of computer. This is especially prevalent in computer labs. Yes, having every machine be exactly the same simplifies setup, troubleshooting, etc. etc. But this is not the world in which your customers live!

This holds true even if you're writing line of business applications for a corporation where the computing environment is tightly controlled and locked down. That "standard" configuration will change on a regular basis, and it will take months or years to complete the switch over. (By which time the standard of course has moved on!)

So be sure to sprinkle the following throughout your development organization (by which term I include dev, test, PM, docs, and anyone else that helps create your app):

- Processors from multiple vendors (i.e., Intel and AMD if you're building Windows applications)
- Multiple versions of each brand of processor (e.g., for Intel, mobile and desktop Celerons and Pentiums)
- Single-, dual-, and multi-processors
- Single-, dual, and multi-core processors
- Hyperthreaded and non-hyperthreaded processors
- Desktop and laptop configurations
- 32-bit and 64-bit configurations

Hardware Configurations

You are not done testing yet unless…you have consciously decided which of the following hardware configurations to include in and which to exclude from your test matrix:

- Low end desktop
- High end desktop
- Low end laptop
- High end laptop

- Minimum expected screen resolution (this may be 640x480 in some cases, but depending on your customers you may be able to expect a higher resolution)
- Super-high screen resolution (yes this is an excuse to get one of those giant flat panels!)
- Other relevant form factors (e.g., for Microsoft Windows: convertible Tablet PC, slate Tablet PC, UMPC/Origami, Pocket PC, Smartphone)
- Maxed-out super-high-end-everything machine
- Minimum configuration machine
- Laptop with power management settings disabled
- Laptop with power management settings set to maximize power
- Laptop with power management settings set to maximize battery life
- CPU configurations

The exact definition of "low end" and "high end" and such will vary across applications and use cases and user scenarios - the minimum configuration for a high-end CAD program will probably be rather different than that for a mainstream consumer-focused house design application, for example. Also carefully consider which chipsets, CPUs, system manufacturers, and such you need to cover. The full matrix is probably rather larger than you have time or budget to handle!

Security

You are not done testing unless...you have thought hard about security testing and made explicit decisions about which testing to do and to not do. Back in the day, when even corporate computers were unlikely to be connected to a network, security testing didn't seem that big of a deal. After all, even if a computer did get infected by a virus it couldn't do much damage! Nowadays, viruses and worms take down corporations' mail systems, even my mother is inundated by spam, and hordes of unknowing consumers host trojan applications doing who knows what under the service of who knows whom. Security testing is now officially a Big Deal. Here are but a few of the plethora of test cases to consider; for more details consult the many big thick security tomes for sale at your favorite bookseller.

- Pore through your source code, APIs, and user interface looking for potential
    - Buffer overrun attacks
    - Denial of service attacks
    - SQL injection attacks
    - Virus attacks
    - User privacy violations (e.g., including user identifying data in saved files)
- On Microsoft Windows OSs, use Application Verifier to ensure no NULL DACLS are created or used - and to check for many other potential security issues
- Verify security for links and macros is sufficient and works correctly
- Verify relative filenames (e.g., "..\..\file") are handled correctly
- Verify temporary files are created in appropriate locations and have appropriate permissions
- Verify your application functions correctly under different user rights and roles
- Verify your application functions correctly under partial trust scenarios
- Verify every input is bounds-checked

- Verify known attack vectors are disabled

Dates and Y2K

You are not done testing unless...you have vetted your application for Year 2000 issues. Even though we are now well past that date, Y2K issues can crop up with the least provocation. Those of you on some form of Unix (e.g., all you Apple users) have another Y2K-ish situation coming up in 2038 when that platform's 32-bit time data structure rolls over. Oh, and as long as you're looking at date-related functionality, you may as well look for other date-related defects as well, such as general leap year handling.

- Verify dates entered with a two digit year from 1 Jan 00 through 31 Dec 29 are interpreted as 1 Jan 2000 through 31 Dec 2029
- Verify dates entered with a two digit year from 1 Jan 30 through 31 Dec 99 are interpreted as 1 Jan 1930 through 31 Dec 1999
- Very dates at least through 2035 are supported
- Verify dates in leap years are correctly interpreted:
- 29 Feb 1900 should fail
- 29 Feb 1996 should work
- 29 Feb 2000 should work
- 31 Dec 2000 should work and be identified as day 366
- 29 Feb 2001 should fail
- Verify other interesting dates are correctly interpreted and represented, including:
- 31 Dec 1999 should work
- 1 Jan 2000 should be unambiguously represented
- 10 Jan 2000 (first seven digit date)
- 10 Oct 2000 (first eight digit date)
- Verify entering "13" for the month in year 2000 fails

Performance and Stress

You are not done testing unless...you understand the performance characteristics of your application and the manner in which your product deforms under stress. Performance testing can seem straightforward: verify the times required to complete typical user scenarios are acceptable - what's so hard about that? Simulating those scenarios sufficiently realistically can be difficult, however. Even more so when it comes to stress testing! For example, say you are testing a web site which you expect to become wildly popular. How do you simulate millions of users hitting your site simultaneously?

I do not have any easy answers for these questions. I do however have several scenarios and conditions to consider:

Performance

- Verify performance tests exist for each performance scenario, and are being executed on a sufficiently regular basis

- Verify performance targets exist for each performance scenario, and are being met
- Verify the performance tests are targeting the correct scenarios and data points
- Verify performance optimizations have the intended effect
- Verify performance with and without various options enabled, such as Clear Type and menu animations, as appropriate
- Compare performance to previous versions
- Compare performance to similar applications

Stress

- Run under low memory conditions
- Run under low disk space conditions
- Run under out-of-memory caused via automation (e.g., a use-up-all-available-memory utility)
- Run under out-of-memory caused by real world scenarios (e.g., running multiple other applications each having multiple documents open)
- Run under a heavy user load
- Run over a network which frequently drops out
- Run over a network with a large amount of traffic
- Run over a network with low bandwidth
- Run on a minimum requirements machine
- Open, save, and execute (as appropriate) from floppies and other removable disks

As you do all of this performance and stress testing, also check for memory and other resource leaks.

Application Configuration and Interoperability

You are not done testing unless…you have given your application's configurability options a thorough going-over. Applications can be configured via many different avenues: global and per-user configuration files and global and per-user registry settings, for example. Users tend to appreciate being able to customize an application to look and work exactly the ways they want it to look and work; they tend to get grumpy if all those customizations are nowhere to be seen the next time they launch the application. Interoperability with other instances of the application and with other applications fall into this same boat: most users expect a certain level of interoperability between applications, and they often get grumpy if your application does not meet that bar. Window interactions show up here too. Here are a few items to kick off your configuration and interoperability brain storming session:

Configuration

- Verify settings which should modify behavior do
- Verify settings are or are not available for administrators to set via policies, as appropriate
- Verify user-specific settings roam with the user
- Verify registry keys are set correctly, and that no other registry keys are modified
- Verify user-specific configuration settings are not written to machine or global registry settings or configuration files

- Brainstorm how backward compatibility problems might occur because a setting has moved or changed and thus broken functionality in a previous version or changed default values from a previous version

Interoperability

- Verify clipboard cut, copy, and paste operations within your application
- Verify clipboard cut, copy, and paste operations between your and other applications
- Verify drag and drop operations within your application
- Verify drag and drop operations between your and other applications

Window Interactions

- Verify z-order works correctly, especially with respect to Always On Top windows (e.g., online help)
- Verify all modal dialog boxes block access to the rest of the application, and all modeless dialog boxes do not
- Verify window and dialog box focus is correct in all scenarios
- Verify window size is correct after restore-from-minimize and restore-from-maximize
- Verify window size is correct on first launch
- Verify window size is correct on subsequent launches
- Verify window size is maintained after a manual resize
- Verify multiple window (i.e., MDI) scenarios work correctly
- Verify window arrangement commands (e.g., Cascade All) work correctly
- Verify multiple instances of the application work correctly across all scenarios (e.g., that a modal dialog box in one instance of the application does not disable interactivity in another instance)

Setup

You are not done testing yet unless...you have tested your program's setup process under the following conditions. Although some of these terms are specific to Microsoft Windows other operating systems generally have similar concepts.

- Installing from a CD-ROM/DVD-ROM
- Installing from a network share
- Installing from a local hard drive
- Installing to a network share
- Installing from an advertised install, where icons and other launch points for the application are created (i.e., the app is "advertised" to the user), but the application isn't actually installed until the first time the user launches the program. Also known as "install on demand" or "install on first use".
- Unattended installs (so-called because no user intervention is required to e.g., answer message boxes), aka command line installs. This can become quite complicated, as the OS's installation mechanism supports multiple command-line options, and your application may support yet more. Oh the combinatorial testing fun!

- Mass installs, via an enterprise deployment process such as Microsoft Systems Management Server.
- Upgrading from previous versions. This can also become quite complicated depending on how many versions of your app you have shipped and from which of those you support upgrades. If all of your customers always upgrade right away, then you're in good shape. But if you have customers on five or six previous versions, plus various service packs and hotfixes, you have a chore ahead of you!
- Uninstall. Be sure that not only are all application-specific and shared files removed, but that registry and other configuration changes are undone as well. Verify components which are shared with other applications are/not uninstalled depending whether any of the sharing apps are still installed. Try out-of-order uninstalls: install app A and then app B, then uninstall app A and then uninstall app B.
- Reinstall after uninstalling the new and previous versions of your application
- Installing on all supported operating systems and SKUs. For Microsoft Windows applications, this may mean going as far back as Windows 95; for Linux apps, consider which distros you will be supporting.
- Minimum, Typical, Full, and Custom installs. Verify that each installs the correct files, enables the correct functionality, and sets the correct registry and configuration settings. Also try upgrading/downgrading between these types - from a minimum to complete install, for example, or remove a feature - and verify that the correct files etc are un/installed and functionality is correctly dis/enabled.
- Install Locally, Run From Network, Install On First Use, and Not Available installs. Depending on how the setup was created, a custom install may allow the individual components to be installed locally, or to be run from a shared network location, or to be installed on demand, or to not be installed at all (I use that one a lot for clip art). Verify that each component supports the correct install types - your application's core probably shouldn't support Not Available, for example. Mix-and-match install types - if you install one component locally, run another from the network, and set a third to Install on First Use, does everything work correctly?
- Install On First Use installs. Check whether components are installed when they need to be (and not before), and that they are installed to the correct location (what happens if the destination folder has been deleted?), and that they get registered correctly.
- Run From Network installs. Check whether your app actually runs - some apps won't, especially if the network share is read-only. What happens if the network is unavailable when you try to launch your app? What happens if the network goes down while the application is running?
- Verify installs to deeply nested folder structures work correctly.
- Verify that all checks made by the installer (e.g., for sufficient disk space) work correctly.
- Verify that all errors handled by the installer (e.g., for insufficient disk space) work correctly.
- Verify that "normal" or limited-access (i.e., non-admin) users can run the application when it was installed by an administrator. Especially likely to be troublesome here are Install On First Use scenarios.

- Verify the application works correctly under remoted (e.g., Microsoft Remote Desktop or Terminal Server), and virtual (e.g., Microsoft Virtual PC and Virtual Server) scenarios. Graphics apps tend to struggle in these cases; I've seen apps just plain refuse to boot when running under Terminal Server.
- Perform a Typical install followed by a Modify operation to add additional features.
- Perform a Custom install followed by a Modify operation to remove features.
- Perform a Typical install, delete one or more of the installed files, then perform a Repair operation.
- Perform a Custom installation that includes non-Typical features, delete one or more of the installed files, then perform a Repair operation.
- Patch previous versions. Patching is different from an upgrade in that an upgrade typically replaces all of the application's installed files, whereas a patch usually overwrites only a few files.
- Perform a Minor Upgrade on a previously patched version.
- Patch on a previously upgraded version.
- Upgrade a previously installed-then-modified install.
- Patch a previously installed-then-modified install.

Setup Special Cases

Beyond the standard setup cases above, also consider some more specialized conditions. As before, although some of these terms are specific to Microsoft Windows other operating systems generally have similar concepts.

Local Caching

Depending how the setup program was authored, it may allow setup files to be cached on the local hard drive, which speeds up subsequent repair and other setup operations.

- Verify the correct files/archives are cached
- Verify all files shared with another feature or application are handled correctly across installs, uninstalls, and reinstalls
- Verify setups for multiple programs and multiple versions of individual programs share the cache correctly. This is especially important for shared files - imagine the havoc that might ensue if uninstalling one application removed from the cache a shared file other installed applications require!
- Verify deleting a file/archive from the cache causes a reinstall and recaches the file/archive

Setup Authoring

Also knows as: Test your setup program.

- Verify every possible path through the setup program (including canceling at every cancel point) works correctly
- Verify the setup program includes the correct components, files, and registry settings

- Verify any custom actions or conditions, creation of shortcuts, and other special authoring works correctly
- Verify the correct install states are available
- Verify canceling an in-progress install in fact cancels and leaves no trace of the unfinished install

Multi-User Setup

What happens when multiple users mess with modify the setup configuration of your application?

- Verify your application works correctly for User2 after User1 installs/modifies/damages it
- Verify per-user features must be installed by User2 even after User1 has installed them
- Verify User2's per-user settings do not change when User1 changes them

Network Setup

Can you install your app from the network rather than a local CD?

- Verify your feature uninstalls cleanly and correctly when it was installed from the network (sometimes called a post-admin install)
- Verify the correct files are installed a) on the network share, and b) locally (as appropriate), when the application is installed as Run From Network

Upgrades

You are not done testing unless...you understand how your application handles being installed over previous versions of your application, and having the operating system upgraded out from under it. You may want to test installing a previous version over, or side-by-side to, the current version as well. Consider whether to cover all three combinations: upgrading just your application, upgrading just your operating system, and upgrading both the operating system and your application.

Application Upgrade

- Verify upgrading over a previous version replaces appropriate files and no others
- Verify installing this version side-by-side to previous versions works correctly
- Verify the correct files do and do not exist after an upgrade, and that their versions are also correct
- Verify default settings are correct
- Verify previously existing settings and files are maintained or modified, as appropriate
- Verify all functionality works correctly when the previous version(s) and/or the new version is set to Run From Network
- Verify any features and applications dependent on files or functionality affected by the upgrade work correctly

Operating System Upgrade

- Verify upgrading over a previous version replaces appropriate files and no others
- Verify all functionality works correctly

- Verify any features and applications dependent on operating system files or functionality affected by the upgrade work correctly

Documentation

You are not done testing unless...you have reviewed all documentation, a) to ensure that it is correct, and b) to help generate test cases. I have lost track of the number of help sample code listings which did not compile due to one problem or another. I have seen documentation which depicted UI which was not in the actual application, and encountered UI in the application which was nowhere to be found in the documentation. Other collateral can be useful to review as well - all those product support calls for the previous version of your application, for example. And have you looked at your source code lately? Source code reviews are a simple way to find those places where supposed-to-be-temporary message boxes and other functionality is about to be shipped to paying customers. And on and on and on.

- Review postponed and otherwise not-fixed bugs from previous releases
- Review product support issues from previous releases
- Review error reports submitted by customers for previous releases
- Verify each error message which can be presented to your customer is accurate, easily understood, and understandable
- Verify each input validation error message refers to the correct problem
- Verify all tutorials are correct: the steps are correct, UI matches the actual UI, and so on
- Review every help topic for technical accuracy
- Verify each piece of context sensitive help is correct
- Verify every code sample functions correctly
- Verify every code sample follows appropriate coding standards and guidelines
- Review all source code for
    - Correctness
    - Lines of code which have not been executed by a test case
    - Security issues (see the Security YANDY list for more details)
    - Potential memory leaks
    - Dead code
    - Correct error handling
    - Use of obsolete and banned function calls
    - Compliance with appropriate coding standards and guidelines
    - Inappropriate user-facing messages
- Verify you have discussed the feature design and target users with your feature team
- Verify you have asked your developer which areas they think could use special attention
- Verify you have discussed your feature with your product support team
- Verify you have brainstormed and reviewed your test cases with your feature team and with your Test team
- Verify you have discussed cross-feature implications with your feature team and with your Test team

- Verify you have completed all appropriate feature-specific testing
- Verify you have completed all appropriate cross-feature integration testing
- Verify you have completed all appropriate real-world use-it-the-way-your-user-will testing

You Are Not Done Yet: Developer Edition

My You Are Not Done Yet list tends to used by testers more than developers, whose eyes seem to glaze over as they start to comprehend its size and detail. One of my missions in life is helping developers learn how to test their code. I have worked with a lot of developers over the years, and I've found that they generally do want to test their code and simply don't know how to go about doing so effectively.

Testers use my You Are Not Done Yet checklist as a final check before they decide that they are done (enough) with their testing. Similarly, this You Are Not Done Yet: Developer Edition checklist is for developers to use as a final check before they declare their code done (enough) to hand over to their testers.

- Customize This List: If you get a bug, determine what test (or even better, what general class of tests or testing technique) would have caught the bug, then add it to this list.
- Use Your Tester: Brainstorm tests with your tester. Review your (planned/actual) tests with your feature team. Coordinate your testing with your tester, especially with respect to tests they have already written/are currently writing.
- Focus On Your Customer: Think, "Where would the presence of bugs hurt our customers the most?", then let your answers drive your testing.
- Test Around Your Change. Consider what it might affect beyond its immediate intended target. Think about related functionality that might have similar issues. If fixing these surrounding problems is not relevant to your change, log bugs for them. To quote a smart person I know, "Don't just scratch it. What's the bug bite telling you?"
- Use Code Coverage. Code coverage can tell you what functionality has not yet been tested. Don't however just write a test case to hit the code. Instead, let it help you determine what classes of testing and test cases that uncovered code indicates you are missing.
- Consider Testability. Hopefully you have considered testability throughout your design and implementation process. If not, think about what someone else will have to do to test your code. What can you do/do you need to do in order to allow proper, authorized verification? (Hint: Test Driven Design is a great way to achieve testable code right from the get-go!)
- Ways To Find Common Bugs:
    - Reset to default values after testing other values (e.g., pairwise tests, boundary condition tests)
    - Look for hard coded data (e.g., "c:\temp" rather than using system APIs to retrieve the temporary folder), run the application from unusual locations, open documents from and save to unusual locations
    - Run under different locales and language packs
    - Run under different accessibility schemes (e.g., large fonts, high contrast)
    - Save/Close/Reopen after any edit
    - Undo, Redo after any edit
- Test Boundary Conditions: Determine the boundary conditions and equivalency classes, and then test just below, at, in the middle of, and just above each condition. If multiple data types

can be used, repeat this for each option (even if your change is to handle a specific type). For numbers, common boundaries include:

- smallest valid value
- at, just below, and just above the smallest possible value
- -1
- 0
- 1
- some
- many
- at, just below, and just above the largest possible value
- largest valid value
- invalid values
- different-but-similar datatypes (e.g., unsigned values where signed values are expected)
- for objects, remember to test with null and invalid instances

- Other Helpful Techniques:
  - Do a variety of smallish pairwise tests to mix-and-match parameters, boundary conditions, etc. One axis that often brings results is testing both before and after resetting to default values.
  - Repeat the same action over and over and over, both doing exactly the same thing and changing things up.
  - Verify that every last bit of functionality you have implemented is discussed in the spec and matches what the spec describes should happen. Then look past the spec and think about what is not happening and should.
  - "But a user would never do that!": To quote Jerry Weinberg, When a developer says, "a user would never do that," I say, "Okay, then it won't be a problem to any user if you write a little code to catch that circumstance and stop some user from doing it by accident, giving a clear message of what happened and why it can't be done." If it doesn't make sense to do it, no user will ever complain about being stopped.