

# Lecture 13: Toolkits

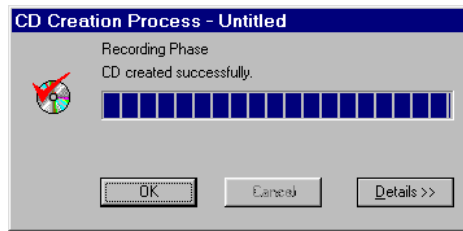
---

Fall 2005

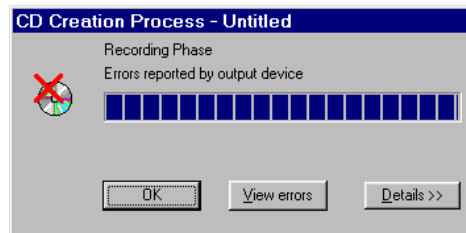
6.831 UI Design and Implementation

1

## UI Hall of Fame or Shame?



Source: UI Hall of Shame



Fall 2005

6.831 UI Design and Implementation

2

Our Hall of Shame candidate for the day is this dialog box from Adaptec Easy CD Creator, which appears at the end of burning a CD. The top image shows the dialog when the CD was burned successfully; the bottom image shows what it looks like when there was an error.

The key problem is the **lack of contrast** between these two states. Success or failure of CD burning is important enough to the user that it should be obvious at a glance. But these two dialogs look identical at a glance. How can we tell? Use the **squint test**, which we talked about in the graphic design lecture. When you're squinting, you see some labels, a big filled progress bar, a roundish icon with a blob of red, and three buttons. All the details, particularly the text of the messages and the exact shapes of the red blob, are fuzzed out. This simulates what a user would see at a quick glance, and it shows that the graphic design doesn't convey the contrast.

One improvement would change the check mark to another color, say green or black. Using red for OK seems **inconsistent** with the real world, anyway. But designs that differ only in red and green wouldn't pass the squint test for color-blind users.

Another improvement might remove the completed progress bar from the error dialog, perhaps replacing it with a big white text box containing a more detailed description of the problem. That would clearly pass the squint test, and make errors much more noticeable.

## UI Hall of Fame or Shame?

	Primary Sort Option	Second Sort Option	Third Sort Option
Part ID	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>
Description	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Vendor ID	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Vendor Number	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Location	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>
Class ID	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
User def 1	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
User def 2	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
User def 3	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Source: UI Hall of Shame

Fall 2005

6.831 UI Design and Implementation

3

Our second Hall of Shame candidate for the day is this interface for choosing how a list of database records should be sorted. Like other sorting interfaces, this one allows the user to select more than one field to sort on, so that if two records are equal on the primary sort key, they are next compared by the secondary sort key, and so on.

On the plus side, this interface communicates one aspect of its model very well. Each column is a set of radio buttons, clearly grouped together (both by **gestalt proximity** and by an explicit raised border). Radio buttons have the property that only one can be selected at a time. So the interface has a clear **affordance** for picking only one field for each sort key.

But, on the down side, the radio buttons don't afford making NO choice. What if I want to sort by only one key? I have to resort to a trick, like setting all three sort keys to the same field. The interface model clearly doesn't map correctly to the task it's intended to perform. In fact, unlike typical model mismatch problems, both the user and the system have to adjust to this silly interface model – the user by selecting the same field more than once, and the system by detecting redundant selections in order to avoid doing unnecessary sorts.

The interface also fails on **minimalist** design grounds. It wastes a huge amount of screen real estate on a two-dimensional matrix, which doesn't convey enough information to merit the cost. The column labels are similarly redundant; "sort option" could be factored out to a higher level heading. The term "Primary" is not really consistent with "Second" and "Third"; "First" would be simpler.

## Today's Topics

---

- Widgets
- Toolkit layering
- Look-and-feel

## Widgets

---

- Reusable user interface components
  - Also called controls, interactors, gizmos, gadgets
- Widget is a view + controller
  - Embedded model
    - Application data must be copied into the widget
    - Changes must be copied out'
  - Linked model
    - Application provides model satisfying an interface
    - Enables "data-bound" widgets, e.g. a table showing thousands of database rows, or a combo box with thousands of choices

Fall 2005

6.831 UI Design and Implementation

5

**Widgets** are the last part of user interface toolkits we'll look at. Widgets are a success story for user interface software, and for object-oriented programming in general. Many GUI applications derive substantial reuse from widgets in a toolkit.

Widgets generally combine a view and a controller into a single tightly-coupled object. For the widget's model, however, there are two common approaches. One is to fuse the model into the widget as well, making it a little MVC complex. With this embedded model approach, application data must be copied into the widget to initialize it. When the user interacts with the widget, the user's changes or selections must be copied back out.

The other alternative is to leave the model separate from the widget, with a well-defined interface that the application can implement.

Embedded models are usually easier for the developer to understand and use for simple interfaces, but suffer from serious scaling problems. For example, suppose you want to use a table widget to show the contents of a database. If the table widget had an embedded model, you would have to fetch the entire database and load it into the table widget, which may be prohibitively slow and memory-intensive. Furthermore, most of this is wasted work, since the user can only see a few rows of the table at a time. With a well-designed linked model, the table widget will only request as much of the data as it needs to display.

The linked model idea is also called **data binding**.

## Widget Pros and Cons

---

- Advantages
  - Reuse of development effort
    - Coding, testing, debugging, maintenance
    - Iteration and evaluation
  - External consistency
- Disadvantages
  - Constrain designer's thinking
  - Encourage menu & forms style, rather than richer direct manipulation style
  - May be used inappropriately

Fall 2005

6.831 UI Design and Implementation

6

Widget reuse is beneficial in two ways, actually. First are the conventional software engineering benefits of reusing code, like shorter development time and greater reliability. A widget encapsulates a lot of effort that somebody else has already put in.

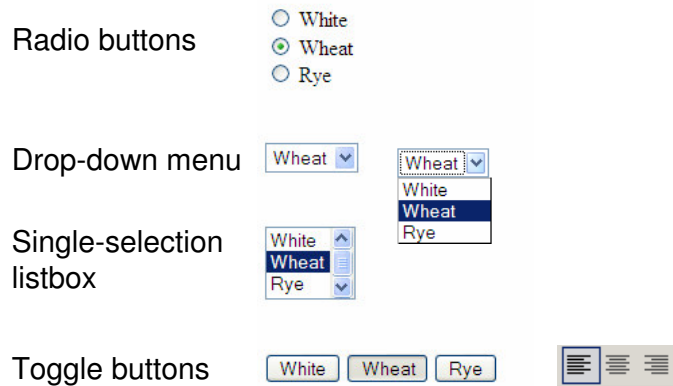
Second are usability benefits. Widget reuse increases consistency among the applications on a platform. It also (potentially) represents *usability* effort that its designers have put into it. A scrollbar's affordances and behavior have been carefully designed, and hopefully evaluated. By reusing the scrollbar widget, you don't have to do that work yourself.

One problem with widgets is that they constrain your thinking. If you try to design an interface using a GUI builder – with a palette limited to standard widgets – you may produce a clunkier, more complex interface than you would if you sat down with paper and pencil and allowed yourself to think freely. A related problem is that most widget sets consist mostly of form-style widgets: text fields, labels, checkboxes – which leads a designer to think in terms of menu/form style interfaces. There are few widgets that support direct visual representations of application objects, because those representations are so application-dependent. So if you think too much in terms of widgets, you may miss the possibilities of direct manipulation.

Finally, widgets can be abused, applied to UI problems for which they aren't suited. We saw an example in Lecture 1 where a scrollbar was used for selection, rather than scrolling.

A great book with tips about how to use common widgets is *GUI Bloopers: Don'ts and Dos for Software Developers and Web Designers*, by Jeff Johnson (Morgan Kaufmann, 2000).

## Widgets for 1-of-N Choices



Fall 2005

6.831 UI Design and Implementation

7

Let's look at some of the ways widgets can be used to solve common interface problems. We'll start by considering **choices**: asking the user to choose from a set of fixed options.

Suppose we're designing the interface for an on-campus sandwich delivery service, and we want to ask what kind of bread the user wants: white, wheat, or rye. This is a **1-of-N choice**, where in this case  $N=3$ . Here are the various ways to do it with familiar widgets:

**Radio buttons** are named after the station preset buttons in old car radios. Pushing one button in popped out all the others, using a mechanical linkage. (Radio buttons are sometimes called option buttons.) Radio buttons are the conventional idiom for making 1-of-N selections. They're fast (all the choices are visible and only one point-and-click makes the choice), but use a lot of screen real estate as  $N$  gets large. If we wanted to ask which MIT building the sandwich should be delivered to, we probably wouldn't want to use radio buttons –  $N$  is too big.

**Toggle buttons** look like ordinary buttons, but if you push them, they toggle back and forth between depressed and undepressed states. In Java, you can get a set of toggle buttons to behave like radio buttons just by adding them to a `ButtonGroup`. Toggle buttons can offer larger targets than radio buttons (Fitts's Law), or tiny targets if screen real estate is at a premium (using icons instead of text labels). Toggle buttons are often used for mode switches, or for 1-of-N choices in toolbars.

A **drop-down menu** shows the current selection, with an arrow next to it for pulling down the list of  $N$  choices to choose from. A **single-selection listbox** is a static version of the same thing, where the list of  $N$  choices is always visible, and the chosen item is indicated by a highlight. Drop-down menus are very compact, but require two pointing operations (point and click to open the list, then point to the new choice), and possibly scrolling if it's a long list. Drop-down menus and listboxes often support keyboard access, however – you can type the first character (or even a prefix) of your choice, and it will jump to it.

All these widgets are useful for 1-of-N choices, but one problem in many toolkits is that they have radically different APIs – so once you've decided on a widget, changing that decision might affect all the code that interacts with it. In Java, for example, even similar widgets like `JList` (for single-selection listboxes) and `JComboBox` (for drop-down menus) have completely different interfaces for providing the list of choices and getting and setting the selected item. HTML is a little better -- the `<select>` element can generate either a drop-down menu or a listbox using the same code (although web browsers conventionally *only* give you a drop-down menu for a 1-of-N choice, reserving listboxes for K-of-N choices). But if you want  $N$  radio buttons instead, you have to radically change your HTML, into  $N$  `<input type=radio>` elements.

Avoid using  $N$  checkboxes for 1-of-N choices. Radiobuttons are a visually distinct widget that clearly communicates that the choices are exclusive (i.e., that you can only select one at a time). Checkboxes fail to convey this – so that few people probably realize that the Superscript and Subscript checkboxes in the typical font dialogs are actually exclusive options.

## Widgets for 1-of-2 Choices

- Widgets for 1-of-N choices (with  $N=2$ ), plus:

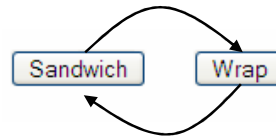
Checkbox  Toasted

Toggle button  Toasted

- Avoid:

Sort in ascending order

Insert vs. overstrike



For the special case where  $N=2$ , you can also use a single **checkbox** or **toggle button**. Only do this if it's a boolean choice, where the alternative (when the checkbox is unchecked or the toggle button is undepressed) is actually a *negation*. The checkboxes on the bottom show some of the nasty ambiguities that can result.

One mistake often seen here is using an ordinary push button for a 1-of-2 choice, where the *label* of the button is changed when the user pushes it. This raises serious confusion about what the label means: is it the *current state* (am I getting a sandwich right now), or the *future state* (should I push this button to get a sandwich)? Even worse would be using this kind of idiom for a 1-of-N choice, where clicking the button cycles through all the choices. There's no excuse for that kind of interface, given the rich set of widgets available for these kinds of choices.



## Widgets for K-of-N Choices

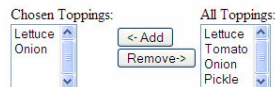
N checkboxes

 Lettuce  
 Tomato  
 Onion  
 Pickle

Multiple-selection  
listbox



Two listboxes



Fall 2005

6.831 UI Design and Implementation

9

Here are widgets commonly used for choose any number of items from a set of N fixed choices.

**Checkboxes** are a natural way to do this.

A **multiple-selection listbox** is another way to do it. (Most listbox widgets have a property you can set that determines whether it should support multiple selection or not.) This approach has a few problems. First is learnability -- the conventions for multiple selection (click to clear and make one selection, control-click to toggle additional selections, shift-click to select a range) aren't visible or known to all users. Another is errors -- the selection is very fragile. One accidental click can clear a carefully-created selection, and recovering from the error may take a lot of work if the list of choices is long.

**Two listboxes** are still another way: one listbox for the K selected choices, the other listing all N. It uses lots of screen real estate (more than twice as much as a single listbox), may be slower to use (hopping between listboxes and buttons, although often this idiom allows double-clicking to move an item to the other listbox), but on the plus side, it's far less fragile than multiple selection, and the K choices you've made are clearly visible, rather than being distributed throughout the N items. Here's a design question: when the user moves an item over to the K list, should it disappear from the N list? Maybe not disappear, since that would remove visual search landmarks from the N list; but some indication in the N list of which items have already been selected would improve the visibility of system status and reduce errors (like selecting the same item again).

Alas, no toolkit provides a double-listbox like this as a built-in widget, so you generally have to roll your own.

## Widgets for Commands

---

- Menubar
- Toolbar
- Context menu (right-click popup menu)
- Push button
- Hyperlink
  
- Command objects (`javax.swing.Action`)
  - action event handler
  - icons and labels
  - tooltip description
  - enabled state

Fall 2005

6.831 UI Design and Implementation

10

Many widgets are available for commands.

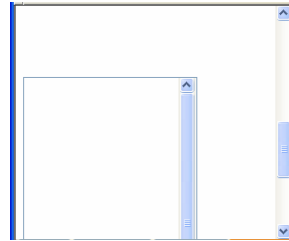
It's a good idea to put the same command in multiple widgets (menubar item, toolbar item, context menu, keyboard shortcut). But it's not a good idea to put it in different *menus* in the menubar, or to use several different *labels* for the same command, because of consistency.

In most toolkits, the command widgets (unlike the choice widgets) actually have a similar API, so you can easily use multiple different widgets for the same application command. In Swing, this interface is the `Action` interface, and it represents (among other things) an event handler that actually performs the command, descriptions of the command (like an icon, a label, and a tooltip), and a boolean flag that indicates whether the command is currently enabled or disabled. Using `Actions` allows you to disable all the widgets that might invoke the command with a single line of code.

In general, don't *remove* command widgets that can't be used at the moment from the UI; just *disable* them. Removing the widgets slows down visual search, because the user might be relying on them as landmarks; it also reduces visibility, since the user can't tell the difference between a command that doesn't exist in the application and one that just isn't available right now.

## Widgets for Window Organization

- 1-of-N panes
  - Tabbed panes
  - Listbox + changing pane
- Multiple content panes
  - Splitters
- Large content panes
  - Scroll panes



Fall 2005

6.831 UI Design and Implementation

11

Most toolkits also have widgets for organizing other widgets and navigating through the organization.

A common problem is grouping an interface into N pages that can be viewed one at a time. **Tabbed panes** are a well-understood idiom for this problem; we've seen in an earlier Hall of Shame, however, that tabs don't scale well, and are largely limited to the number of tabs you can fit into one row. A more scalable solution is a table of contents **listbox** whose single-selection controls which page is visible.

An application window may have several content panes, often separated by **splitters** that allow the user to adjust the share of screen real estate given to each pane. Mail clients like Outlook or Thunderbird, for example, have one pane for the folder list, another for a message list, and another for the current message. One design issue to pay attention to is **keyboard focus** and **selection**. Make sure that the pane with the keyboard focus displays its selection in a way that is visually distinguishable from selections in the other panes. Otherwise, actions like Copy, Paste, or Delete become ambiguous – leading to mode errors.

Content that's too large to fit in the window will require scrollbars – and **scrollpane** widgets fortunately make this easy to handle. Be aware that horizontal scrolling is harder than vertical scrolling, first because mouse scroll wheels don't optimize for it (since vertical scrolling is far more common), and second because reading text with a horizontal scrollbar requires scraping the scrollbar back and forth. If you have a choice, organize your UI for vertical scrolling only.

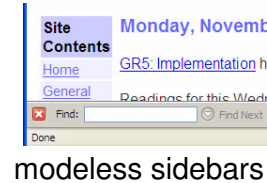
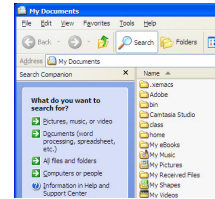
Nested scrollpanes are also problematic – a scrollpane inside another scrollpane -- because the inner scrollbar *can't* bring the inner content fully into view. Text areas and listboxes are often nested inside scrollpanes (like, say, a web browser), and the result may look something like this. Avoid it when possible.

## Widgets for Dialogs

- Modal dialog box
- Modeless dialog box
- Sidebar



modal sheet



modeless sidebars

Fall 2005

6.831 UI Design and Implementation

12

Finally, most toolkits provide widgets for creating dialogs – temporary or long-term satellite windows.

**Modal dialog boxes** are the most common kind of dialog box. They're called *modal* because the application enters a mode in which the only legal actions are interactions with the dialog box – you can't use any other part of the application. On Windows, you can't even *move* the other windows of the application!

The reason is that modal dialog boxes are often implemented on Windows by entering a *new* event loop, which can only handle events for the dialog box; all events for the main window, including the move-window events that the Windows window manager sends it when the user drags the title bar, are simply queued up waiting for the second event loop to return.

This behavior, incidentally, makes modal dialog boxes trivial to use for an application programmer, even when you're writing single-threaded code: you can just call the dialog box as if it were a function, and the whole application stops responding until the user fills it out and dismisses it. It's like a throwback to synchronous I/O programming: the program pops up a prompt (the dialog box) and waits for the user to answer it, with no other user actions allowed.

Modal dialogs do have some usability advantages, such as error prevention (the modal dialog is always on top, so it can't get lost or be ignored, and the user can't accidentally change the selection in the main window while working on a modal dialog that affects that selection) and dialog closure. But there are usability disadvantages too, chief among them loss of user control, reduced visibility (e.g., you can't see important information or previews in the main window), and failures in task analysis might bite you hard (e.g., forcing the user to remember information from one modal dialog to another, rather than viewing both side-by-side).

On Windows, modal dialogs are generally *application-modal* – all windows in the application stop responding until the dialog is dismissed. (The old days of GUIs also had *system-modal* dialogs, which suspended *all* applications.) Mac OS X has a neat improvement, *window-modal* dialogs, which are displayed as translucent sheets attached to the titlebar of the blocked window. This tightly associates the dialog with its window, gives a little visibility of what's underneath it in the main window – and allows you to interact with other windows, even if they're from the same application.

Modeless dialogs don't prevent using other windows. They're often used for ongoing interactions with the main window, like Find/Replace. One problem is that a modeless dialog box can get in the way of viewing or interacting with the main window (as when a Find/Replace dialog covers up the match). Another problem is that there's often no strong visual distinction between a modal dialog and a modeless dialog; sometimes the presence of a Minimize button is a clue, but it's not very strong. (This is another advantage of Mac sheets – the modal sheet is easy to distinguish from a modeless window.)

A modeless dialog may be better represented as a **sidebar**, a temporary pane in the main window that's anchored to one side of the window. Then it can't obscure the user's work, can't get lost, and is clearly visually different from a modal dialog box. The screenshots here show the Find Files sidebar in Windows Explorer, and the Find pane (actually anchored to the bottom of the window, not the side) in Firefox.

## Toolkits

---

- User interface toolkit consists of:
  - Components (view hierarchy)
  - Stroke drawing
  - Pixel model
  - Input handling
  - Widgets

By now, we've looked at all the basic pieces of a user interface toolkit: widgets, view hierarchy, stroke drawing, and input handling. Every modern GUI toolkit provides these pieces in some form. Microsoft Windows, for example, has widgets (e.g., buttons, menus, text boxes), a view hierarchy (consisting of *windows* and *child windows*), a stroke drawing package (GDI), pixel representations (called bitmaps), and input handling (messages sent to a *window procedure*).

## Toolkit Examples

---

	<u>MS Win</u>	<u>Swing</u>	<u>HTML</u>
components	windows	JComponents	elements
strokes	GDI	Graphics	-- (none)
pixels	bitmaps	Image	inlined images
input	messages -> window proc	listeners	Javascript event handlers
widgets	button, menu, textbox, ...	JButton, JMenu, ...	form controls & links

Fall 2005

6.831 UI Design and Implementation

14

Here's a comparison of three UI toolkits: low-level Microsoft Windows (which few people program in anymore); Java Swing; and HTML.

## Toolkit Layering

---



Fall 2005

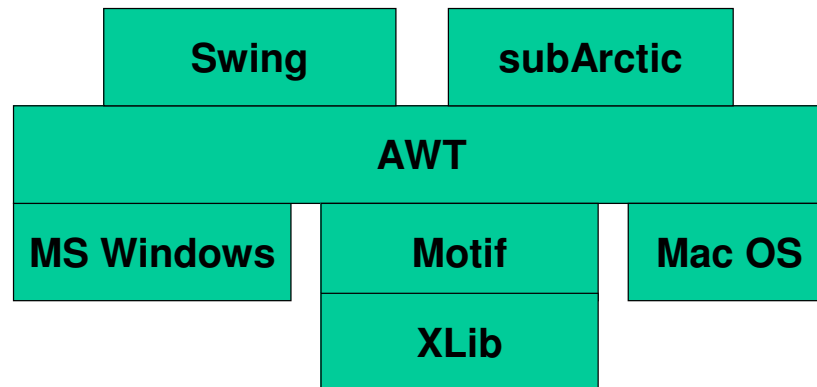
6.831 UI Design and Implementation

15

User interface toolkits are often built on top of other toolkits, sometimes for portability or compatibility across platforms, and sometimes to add more powerful features, like a richer stroke drawing model or different widgets.

X Windows demonstrates this layering technique. The view hierarchy, stroke drawing, and input handling are provided by a low-level toolkit called XLib. But XLib does not provide widgets, so several toolkits are layered on top of XLib to add that functionality: Athena widgets and Motif, among others. More recent X-based toolkits (GTK+ and Qt) not only add widgets to XLib, but also hide XLib's view hierarchy, stroke drawing, and input handling with newer, more powerful models, although these models are implemented internally by calls to XLib.

## Cross-Platform Toolkit Layering



Fall 2005

6.831 UI Design and Implementation

16

Here's what the layering looks like for some common Java user interface toolkits.

AWT (Abstract Window Toolkit, usually pronounced like “ought”) was the first Java toolkit. Although its widget set is rarely used today, AWT continues to provide drawing and input handling to more recent Java toolkits.

Swing is the second-generation Java toolkit, which appeared in the Java API starting in Java 1.2. Swing adds a new view hierarchy (JComponent) derived from AWT's view hierarchy (Component and Container). It also replaces AWT's widget set with new widgets that use the new view hierarchy.

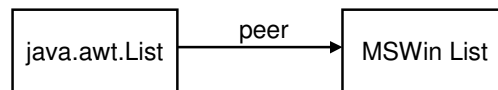
subArctic was a research toolkit developed at Georgia Tech. Like Swing, subArctic relies on AWT for drawing and input handling, but provides its own widgets and views.

Not shown in the picture is SWT, IBM's Standard Widget Toolkit. (Usually pronounced “swit”. Confusingly, the W in SWT means something different from the W in AWT.) Like AWT, SWT is implemented directly on top of the native toolkits. It provides different interfaces for widgets, views, drawing, and input handling.



## Cross-Platform Widgets: AWT Approach

- AWT, HTML
  - Use native widgets, but only those common to all platforms
    - Tree widget available on MS Win but not X, so AWT doesn't provide it
  - Very consistent with other platform apps, because it uses the same code



Fall 2005

6.831 UI Design and Implementation

17

Cross-platform toolkits face a special issue: should the native widgets of each platform be reused by the toolkit? One reason to do so is to preserve consistency with other applications on the same platform, so that applications written for the cross-platform toolkit look and feel like native applications. This is what we've been calling external consistency.

Another problem is that native widgets may not exist for all the widgets the cross-platform toolkit wants to provide. AWT throws up its hands at this problem, providing only the widgets that occur on every platform AWT runs on: e.g., buttons, menus, list boxes, text boxes.

## Cross-Platform Widgets: Swing approach

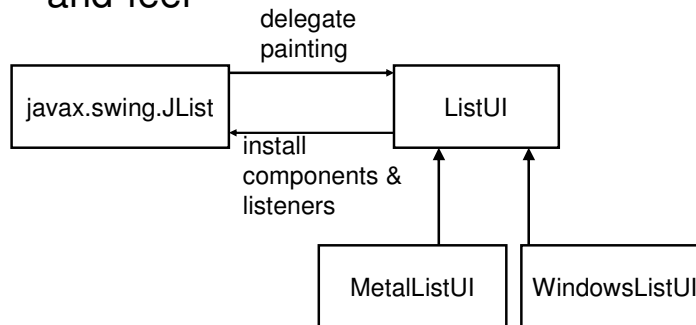
---

- Swing, Amulet
  - Reimplement all widgets
  - Not constrained by least common denominator
  - Consistent behavior for application across platforms

One reason NOT to reuse the native widgets is so that the application looks and behaves consistently with itself across platforms – a variant of internal consistency, if you consider all the instantiations of an application on various platforms as being part of the same system. Cross-platform consistency makes it easier to deliver a well-designed, usable application on all platforms – easier to write documentation and training materials, for example. Java Swing provides this by reimplementing the widget set using its default (“Metal”) look and feel. This essentially creates a Java “platform”, independent of and distinct from the native platform.

## Pluggable Look-and-Feel

- Swing also reimplements platform look-and-feel



Fall 2005

6.831 UI Design and Implementation

19

But Swing also supports external consistency – you can change the appearance and behavior of its widgets to make them resemble the native platform widgets. This is possible because each Swing widget delegates its painting and input handling to a *look and feel object*. Different sets of look-and-feel objects copy the appearance and behavior of different platforms, like Windows, Macintosh, and Motif. Unfortunately there's a lot involved in copying look and feel, and some of these platforms are moving targets – so Swing's Windows look and feel has lagged behind the changes being made in the Windows environment, making Swing applications stand out.

## Cross-Platform Widgets: SWT Approach

---

- SWT
  - Use native widgets where available
  - Reimplement missing native widgets

Recall that AWT (and HTML) only offer widgets that are available on all platforms. SWT takes the opposite tack; instead of limiting itself to the **intersection** of the native widget sets, SWT strives to provide the **union** of the native widget sets. SWT uses a native widget if it's available, but reimplements it if it's missing, attempting to match the "style" of the platform in the new implementation as much as possible.