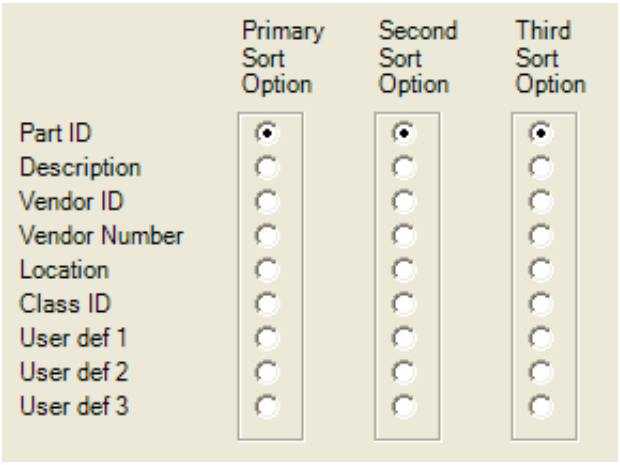
Lecture 13: Toolkits

UI Hall of Fame or Shame?



Our 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.

Today's Topics

- Widgets
- Toolkit layering
- Look-and-feel

Today we'll finish up our survey of user interface implementation.

Last lecture, we discussed the component model, stroke model, and pixel model, and observed that virtually every graphical user interface uses all three of these models for output. The main decision is, at what point in a GUI application do you make the switch from one model to the next. The switch from component model to stroke model occurs at the leaves of the view hierarchy (although parent containers may also draw strokes, of course). The switch from stroke model to pixel model usually occurs within the system's graphics library.

Widgets

- Reusable user interface components
  - Also called controls, interactors, gizmos, gadgets
- Examples
  - Buttons, checkboxes, radio buttons
  - List boxes, combo boxes, drop-downs
  - Menus, toolbars
  - Scrollbars, splitters, zoomers
  - One-line text, multiline text, rich text
  - Trees, tables
  - Simple dialogs

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.

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

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.

Widget Design

- 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

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**.

Toolkits

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

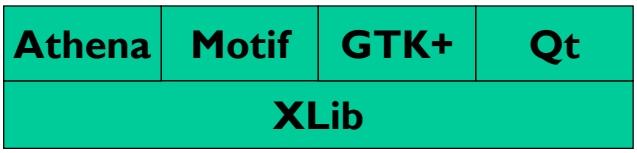
By now, we've looked at all the basic pieces of a user interface toolkit: widgets, view hierarchy, stroke drawing, and input handling. Every modern GUI toolkit provides these pieces in some form. Microsoft Windows, for example, has widgets (e.g., buttons, menus, text boxes), a view hierarchy (consisting 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

	MS Win	Swing	HTML
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

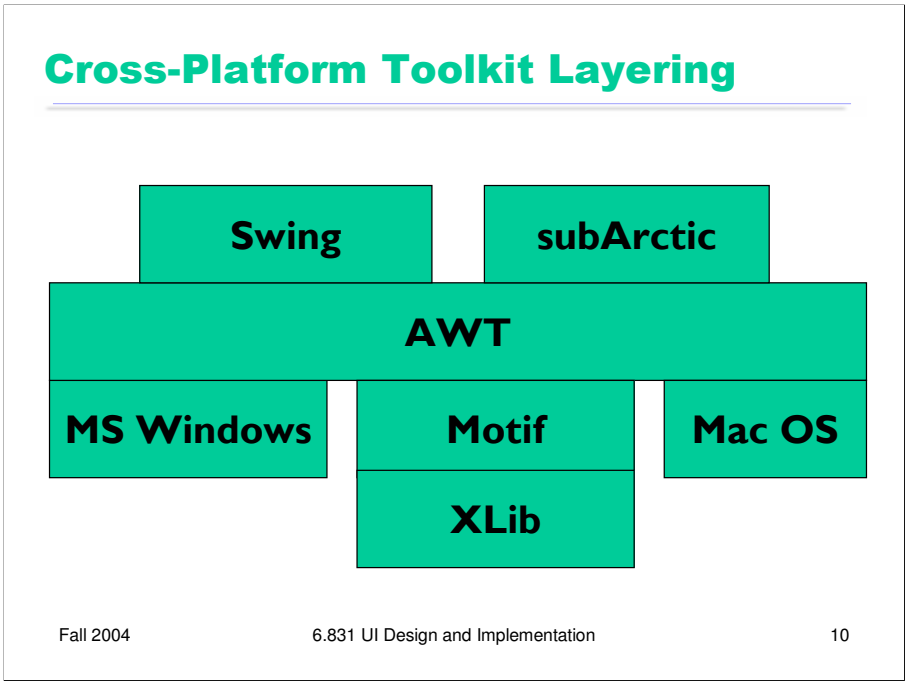
Here's a comparison of three UI toolkits: low-level Microsoft Windows (which few people program in anymore); Java Swing; and HTML.

Toolkit Layering



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.



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.

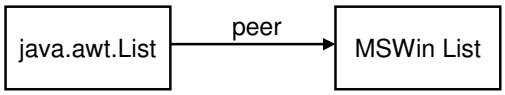
Swing is the second-generation Java toolkit, which appeared in the Java API starting in Java 1.2. Swing adds a new view hierarchy (JComponent) derived from AWT's view hierarchy (Component and Container). It also replaces AWT's widget set with new widgets that use the new view hierarchy.

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



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.

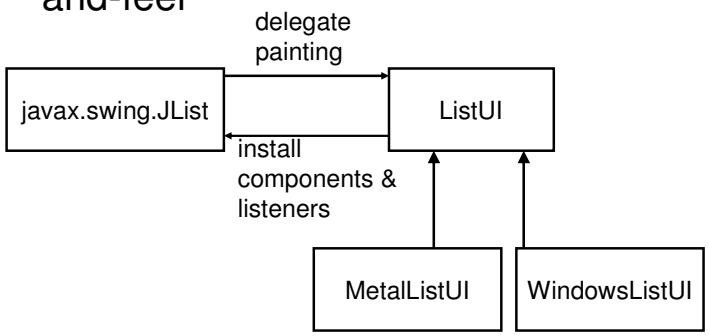
Cross-Platform Widgets: Swing approach

- 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



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.

A Novel Toolkit: Piccolo

- Toolkit for zooming user interfaces
  - Components
    - View hierarchy is actually a **graph**
    - Components can translate, rotate, scale
    - Parents transform but **don't clip** their children by default
  - Strokes
    - Swing Graphics
  - Pixel
    - Swing Images
  - Input
    - BasicInput for mouse and keyboard events
    - Dragging controller
    - Built-in controllers for panning and zooming cameras
  - Widgets
    - PText (multiline text), PImage (images), PPath (shapes)
    - PCamera (viewport), PLayer (composite)

Finally, let's look at Piccolo, a novel UI toolkit developed at University of Maryland. Piccolo is specially designed for building **zoomable** interfaces, which use smooth animated panning and zooming around a large space. We can look at Piccolo in terms of the various aspects we've discussed in this lecture.

Layering: First, Piccolo is a layered toolkit: it runs on top of Java Swing. It also runs on top of .NET, making it a cross-platform toolkit. Piccolo ignores the platform widgets entirely, making no attempt to reimplement or reuse them. (An earlier version of Piccolo, called Jazz, could reuse Swing widgets.)

Components: Piccolo has a view hierarchy consisting of PNode objects. The hierarchy is not merely a tree, but in fact a graph: you can install camera objects in the hierarchy which act as viewports to other parts of the hierarchy, so a component may be seen in more than one place on the screen. Another distinction between Piccolo and other toolkits is that every component has an arbitrary transform relative to its parent's coordinate system – not just translation (which all toolkits provide), but also rotation and scaling. Furthermore, in Piccolo, parents do not clip their children by default. If you want this behavior, you have to request it by inserting a special clipping object (a component) into the hierarchy. As a result, components in Piccolo have two bounding boxes – the bounding box of the node itself (getBounds()), and the bounding box of the node's entire subtree (getFullBounds()).

Strokes: Piccolo uses the Swing Graphics package, augmented with a little information such as the camera and transformations in use.

Pixels: Piccolo uses Swing images for direct pixel representations.

Input: Piccolo has the usual mouse and keyboard input (encapsulated in a single event-handling interface called BasicInput), plus generic controllers for common operations like dragging, panning, and zooming. By default, panning and zooming is attached to any camera you create: dragging with the left mouse button moves the camera view around, and dragging with the right mouse button zooms in and out.

Widgets: the widget set for Piccolo is fairly small by comparison with toolkits like Swing and .NET, probably because Piccolo is a research project with limited resources. It's worth noting, however, that Piccolo provides reusable components for shapes (e.g. lines, rectangles, ellipses, etc), which in other toolkits would require revering to the stroke model.

Piccolo home page: http://www.cs.umd.edu/hcil/piccolo/

Overview: http://www.cs.umd.edu/hcil/piccolo/learn/patterns.shtml

API documentation: http://www.cs.umd.edu/hcil/jazz/learn/piccolo/doc-1.1/api/