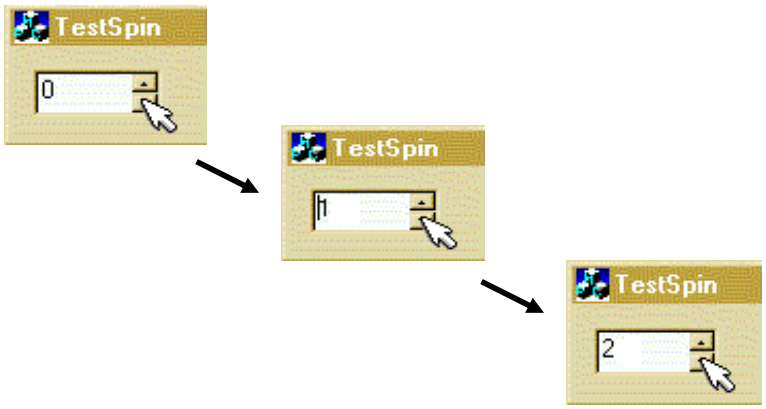
Lecture 9: UI Software Architecture

UI Hall of Fame or Shame?



Source: Interface Hall of Shame

Today's hall of shame candidate is the spin control from Microsoft Visual C++ 5.0. By default, the spin control is configured so that clicking on the down arrow *increments* the value shown in the control. The up arrow decrements the value, so at least the design is consistent with itself, but it isn't consistent with most users' models of how up and down should change a numerical value. To actually use this spin control in an interface, a developer would have to override the default behavior of the buttons and reverse them.

One might speculate that the reason for this strange default behavior is that the spin control was based on code for a *scrollbar*. A spin control is basically a scroll bar with the scroll track and thumb removed, leaving only the up and down arrows. Furthermore, the internal model of a typical scrollbar is an integer representing the scrollbar's position – which usually starts at 0 with the thumb at the top, and increases as the thumb moves down the scrollbar. So the spin control designer may have simply started with a scrollbar, stripped out the unneeded parts, and exposed the internal model as the value of the control. In other words, the **implementation model** has been exposed, incorrectly, as the **interface model**.

Today's Topics

- Computer prototype post-mortem
- Model-view-controller
- View hierarchy

Starting with today's lecture, we'll be talking about how graphical user interfaces are implemented. Today we'll take a high-level look at the software architecture of GUI software, focusing on the **design patterns** that have proven most useful. Two of the most important patterns are the **model-view-controller** abstraction, which has evolved somewhat since its original formulation in the early 80's; and the **view hierarchy**, which is a central feature in the architecture of every popular GUI toolkit.

First, however, we'll discuss the outcome of computer prototyping.

Computer Prototype Post-Mortem

- Storyboard vs. form builder
  - Any hybrid approaches?
- Tools used (survey)
  - Good or bad features?
  - Limitations?
- Tricks?
- How much coding required?
- Could you get the fidelity you wanted?

The class discussed the computer prototyping assignment after the fact. Some of the points raised:

•Roughly 5 groups did storyboards, 10 groups used a form builder.

•Flash was very effective for building hybrid prototypes, since it lets you draw anything but also includes standard widgets like a form builder. HTML imagemaps could be combined with HTML forms in the same way.

•Only one group used PowerPoint for a storyboard, and found it painful because its default gridding prevented aligning screenshot components perfectly. Gridding can be turned off.

•Most of the form builders used were Java-based. JBuilder was widely criticized for bugginess and strange behavior. NetBeans/Forte seemed better.

•Form builder users found themselves debugging Java code even though they didn't write any backend.

Model-View-Controller Pattern

- Separates frontend concerns from backend concerns
- Separates input from output
- Permits multiple views on the same application data
- Permits views/controllers to be reused for other models
- Example: text box
  - Model: mutable string
  - View: rectangle with text drawn in it
  - Controller: keystroke handler

The **model-view-controller** pattern, originally articulated in the Smalltalk-80 user interface, has strongly influenced the design of UI software ever since. In fact, MVC may have single-handedly inspired the software design pattern movement; it figures strongly in the introductory chapter of the seminal "Gang of Four" book (Gamma, Helm, Johnson, Vlissides, *Design Patterns: Elements of Reusable Software*).

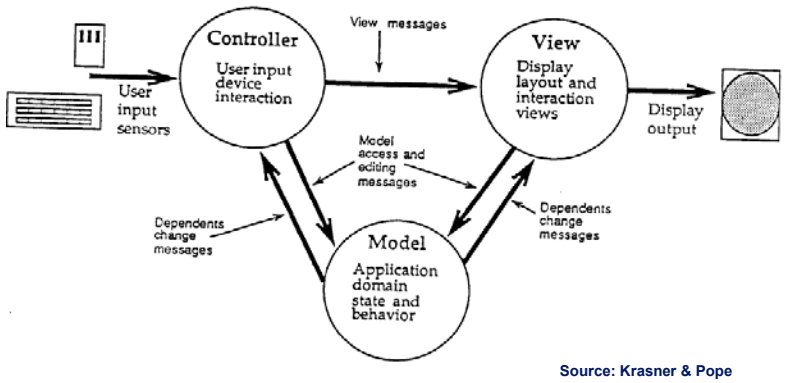
MVC's primary goal is separation of concerns. It separates the user interface frontend from the application backend, by putting backend code into the model and frontend code into the view and controller. MVC also separates input from output; the controller is supposed to handle input, and the view is supposed to handle output.

In principle, this separation has several benefits. First, it allows the interface to have multiple views showing the same application data. For example, a database field might be shown in a table and in an editable form at the same time. Second, it allows views and controllers to be reused for other models, in other applications. The MVC pattern enables the creation of user interface **toolkits**, which are libraries of reusable interface objects.

In practice, the MVC pattern doesn't quite work out the way we'd like. We'll see why.

A simple example of the MVC pattern is a text box widget. Its model is a mutable string of characters. The view is an object that draws the text on the screen (usually with a rectangle around it to indicate that it's an editable text field). The controller is an object that receives keystrokes typed by the user and inserts them in the string.

MVC Diagram



Source: Krasner & Pope

Here's a schematic diagram of the interactions between model, view, and controller. (Figure taken from Krasner & Pope, "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System", *JOOP* v1 n3, 1988).

Model

- Responsible for data
  - Maintains application state (data fields)
  - Implements state-changing behavior
  - Notifies dependent views/controllers when changes occur (observer pattern)
- Design issues
  - How fine-grained are the change descriptions?
    - "The string has changed somehow" vs. "Insertion between offsets 3 and 5"
  - How fine-grained are the observable parts?
    - Entire string vs. only the part visible in a view

Let's look at each part in a little more detail. The model is responsible for maintaining application-specific data and providing access to that data. Models are often mutable, and they provide methods for changing the state safely, preserving its representation invariants.

OK, all mutable objects do that. But a model must also notify its clients when there are changes to its data, so that dependent views can update their displays, and dependent controllers can respond appropriately. Models do this notification using the **observer pattern**, in which interested views and controllers register themselves as listeners for events generated by the model.

Designing these notifications is not always trivial, because a model typically has many parts that might have changed. Even in our simple text box example, the string model has a number of characters. A list box has a list of items. When a model notifies its views about a change, how finely should the change be described? Should it simply say "something has changed", or should it say "these particular parts have changed"? Fine-grained notifications may save dependent views from unnecessarily querying state that hasn't changed, at the cost of more bookkeeping on the model's part.

Fine-grained notifications can be taken a step further by allowing views to make fine-grained registrations, registering interest only in certain parts of the model. Then a view displaying a small portion of a large model would only receive events for changes in the part it's interested in.

Reducing the grain of notification or registration is crucial to achieving good interactive view performance on large models.

View

- Responsible for output
  - Occupies screen extent (position, size)
  - Draws on the screen
  - Listens for changes to the model
  - Queries the model to draw it
- A view has only one model
  - But a model could have many views

In MVC, view objects are responsible for output. A view occupies some chunk of the screen, usually a rectangular area. Basically, the view queries the model for data and draws the data on the screen. It listens for changes from the model so that it can update the screen to reflect those changes.

Controller

- Responsible for input
  - Listens for keyboard & mouse events
  - Instructs the model or the view to change accordingly
    - e.g., keystroke is inserted into the text string
- A controller has only one model and one view

Finally, the controller handles all the input. It receives keyboard and mouse events, and instructs the model to change accordingly. For example, the controller of a text box receives keystrokes and inserts them into the text string.

In the original MVC pattern used in Smalltalk-80, there was only one controller for each model and view.

Problem: Controller Needs Output Too

- Menus are clearly controller-related
  - e.g. right-click menu on a text field
- But a menu needs to be drawn
  - A menu is a model-view-controller in itself, used as a subcomponent

The MVC pattern has a few problems when you try to apply it, which boil down to this: you can't cleanly separate input and output in a graphical user interface. Let's look at a few reasons why.

First, a controller often needs to produce its own output. A good example is a popup menu – in the context of our text box example, this might be the right-click menu that lets you cut, copy, or paste. The menu is clearly part of the controller. Its appearance depends on the controller's state -- e.g., highlighting the menu option that the mouse is hovering over – not strictly on the model's state, like the view does.

Problem: Who Remembers the Selection?

- Must be displayed by the view
  - As blinking text cursor or highlighted object
- Must be updated and used by the controller
  - Clicking or arrow keys change selection
  - Commands modify the model parts that are selected
- Should selection be in model?
  - Generally not
  - Some views need independent selections (e.g. two windows on the same document)
  - Other views need synchronized selections (e.g. table view & chart view)

Second, some pieces of state in a user interface don't have an obvious home in the MVC pattern. One of those pieces is the **selection**. Many UI components have some kind of selection, indicating the parts of the interface that the user wants to use or modify. In our text box example, the selection is either an insertion point or a range of characters.

Which object in the MVC pattern should be responsible for storing and maintaining the selection? The view has to display it, e.g. by highlighting the corresponding characters in the text box. But the controller has to use it and modify it. Keystrokes are inserted into the text box at the location of the selection, and clicking or dragging the mouse or pressing arrow keys changes the selection.

Perhaps the selection should be in the model, like other data that's displayed by the view and modified by the controller? Probably not. Unlike model data, the selection is very transient, and belongs more to the frontend (which is supposed to be the domain of the view and the controller) than to the backend (the model's concern). Furthermore, multiple views of the same model may need independent selections. In Emacs, for example, you can edit the same file buffer in two different windows, each of which has a different cursor.

So we need a place to keep the selection, and similar bits of data representing the transient state of the user interface. It isn't clear where in the MVC pattern this kind of data should go.

Problem: Direct Manipulation

- Direct manipulation: user points at displayed objects and manipulates them directly
- View must provide **affordances** for controller
  - e.g. scrollbar thumb, selection handles
- View must also provide **feedback** about controller state
  - e.g., button is depressed

Here's a third example of why input and output are hard to decouple. Good graphical user interfaces support **direct manipulation**, which means that the user can manipulate displayed objects directly, as if they were physical objects. A scrollbar is a good example of direct manipulation: the user can change the position of the scrollbar thumb by clicking and dragging it directly. Drawing editors provide lots of direct manipulation: you can drag an object to the position you want it, and you can drag the **selection handles** drawn around it to resize the object.

Direct manipulation techniques force a close cooperation between the view and the controller. The view must display **affordances** for manipulation, such as selection handles or scrollbar thumbs. The controller must be aware of the screen locations of these affordances. When the user starts manipulating, the view must modify its appearance to give **feedback** about the manipulation, e.g. painting a button as if it were depressed.

Recall that affordances and feedback were two of Norman's usability principles. The usability principles interact with software design issues here, forcing tighter coupling between two components that, all other things equal, we might prefer to keep separate. But the software design is in the service of the user interface, so usability should take precedence.

Reality: View and Controller are Tightly Coupled

- MVC has largely been superseded by MV (Model-View)
- A view class manages both output and input
  - Selection, affordances, and feedback are managed by the view
  - Mouse and keyboard events are handled also
- Vestiges of controller architecture remain
  - Actions in Java Swing: objects that sit behind menu item, toolbar button, or keyboard shortcut
  - E.g. cut, copy, paste, delete

In principle, it was a nice idea to separate input and output into separate classes. In reality, it isn't feasible, because input and output are tightly coupled. As a result, the MVC pattern has largely been superseded by what might be called Model-View, in which the view and the controller are fused together into a single class. (The term MVC still persists, but people who use it tend to mean something different from the way it was meant in the original Smalltalk design.)

Are there any vestiges of controllers left in modern GUIs? One might argue that any input event handler – like the MouseListener interface in Java – is a controller. But MouseListener is an interface, not a reusable component, and the MouseListener interface tends to be implemented by a view (or by an inner class of a view).

Controllers still exist, but at a higher level than raw mouse and keyboard input. In Java Swing, for example, an Action is a reusable object that represents a command. It sits behind a menu item, toolbar button, or keyboard shortcut, and gets triggered when the user invokes it. Swing actually includes a number of reusable Actions for editing text models: cut, copy, paste, delete, etc.

View Hierarchy

- Views are arranged into a hierarchy
- Containers
  - Window, panel, rich text widget
- Components
  - Canvas, button, label, textbox
  - Containers are also components
- Every GUI system has a view hierarchy, and the hierarchy is used in lots of ways
  - Output
  - Input
  - Layout

The second important pattern we want to discuss in this lecture is the **view hierarchy**.

Views are arranged into a hierarchy of containment, which some views (called containers in the Java nomenclature) can contain other views (called components in Java). A crucial feature of this hierarchy is that containers are themselves components – i.e., Container is a subclass of Component. Thus a container can include other containers, allowing a hierarchy of arbitrary depth.

Virtually every GUI system has some kind of view hierarchy. The view hierarchy is a powerful structuring idea, which is loaded with a variety of responsibilities in a typical GUI. We'll look at three ways the view hierarchy is used: for output, input, and layout.

View Hierarchy: Output

- Drawing
  - Draw requests are passed top-down through the hierarchy
- Clipping
  - Parent container prevents its child components from drawing outside its extent
- Z-order
  - Children are (usually) drawn on top of parents
  - Child order dictates drawing order between siblings
- Coordinate system
  - Every container has its own coordinate system (origin usually at the top left)
  - Child positions are expressed in terms of parent coordinates

First, and probably primarily, the view hierarchy is used to organize output: drawing the views on the screen. **Draw requests** are passed down through the hierarchy. When a container is told to draw itself, it must make sure to pass the draw request down to its children as well.

The view hierarchy also enforces a spatial hierarchy by **clipping** – parent containers preventing their children from drawing anything outside their parent's boundaries.

The hierarchy also imposes an implicit layering of views, called **z-order**. When two components overlap in extent, their z-order determines which one will be drawn on top. The z-order corresponds to an in-order traversal of the hierarchy. In other words, children are drawn on top of their parents, and a child appearing later in the parent's children list is drawn on top of its earlier siblings.

Each component in the view hierarchy has its own coordinate system, with its origin (0,0) usually at the top left of its extent. The positions of a container's children are expressed in terms of the container's coordinate system, rather than in terms of full-screen coordinates. This allows a complex container to move around the screen without changing any of the coordinates of its descendents.

View Hierarchy: Input

- Event dispatch and propagation
  - Raw input events (key presses, mouse movements, mouse clicks) are sent to lowest component
  - Event propagates up the hierarchy until some component handles it
- Keyboard focus
  - One component in the hierarchy has the focus (implicitly, its ancestors do too)

In most GUI systems, the view hierarchy also participates in input handling.

Raw mouse events – button presses, button releases, and movements – are sent to the smallest component (deepest in the view hierarchy) that encloses the mouse position. If this component chooses not to handle the event, it passes it up to its parent container. The event propagates upward through the view hierarchy until a component chooses to handle it, or until it drops off the top, ignored.

Keyboard events are treated similarly, except that the first component to receive the event is determined by the **keyboard focus**, which always points to some component in the view hierarchy.

View Hierarchy: Layout

- Automatic layout: children are positioned and sized within parent
  - Allows window resizing
  - Smoothly deals with internationalization and platform differences (e.g. fonts or widget sizes)
  - Lifts burden of maintaining sizes and positions from the programmer
    - Although actually just raises the level of abstraction, because you still want to get the graphic design (alignment & spacing) right

The view hierarchy is also used to direct the **layout** process, which determines the extents (positions and sizes) of the views in the hierarchy. Many GUI systems have supported automatic layout, including Motif (an important early toolkit for X Windows), Tk (a toolkit developed for the Tcl scripting language), and of course Java AWT.

Automatic layout is most useful because it allows a view hierarchy to adjust itself automatically when the user resizes its window, changing the amount of screen real estate allocated to it. Automatic layout also smoothly handles variation across platforms, such as differences in fonts, or differences in label lengths due to language translation. Finally, it relieves a GUI programmer from the burden of low-level layout programming. Without automatic layout, a typical view class would be full of magic numbers for the positions and sizes of components, and maintaining those numbers would be a nightmare. In practice, though, automatic layout doesn't eliminate the burden of layout; it just raises it to a higher level of abstraction. A good UI developer must learn how to configure automatic layout tools to get the graphic design right, particularly in alignment and spacing.

How Automatic Layout Works

- Components pass size requests up to parent; aggregated up hierarchy
- Parents pass position and size settings down to children, propagating down hierarchy

Whereas the view hierarchy was used in a top-down direction for drawing, and bottom-up for event propagation, automatic layout uses both directions. Here's how a typical automatic layout process works.

First, the components make size requests from their parents. For example, a label would ask for enough space to display itself entirely, which depends on the text of the label and the font used to display it. A container's size request depends on the sizes of its children, so the size requests are aggregated up the hierarchy in a bottom-up manner. (In practice, the *control* for this step is passed top-down, i.e., parents call their children for their sizes. But the *data* is certainly passing bottom-up.)

Once the size requests are known, layout then proceeds top-down, with each container determining the positions and sizes of its children and passing them down the hierarchy.

Next Time

- More patterns commonly seen in UIs
  - Decorator
  - Strategy
  - Flyweight
- Output models
- Input models