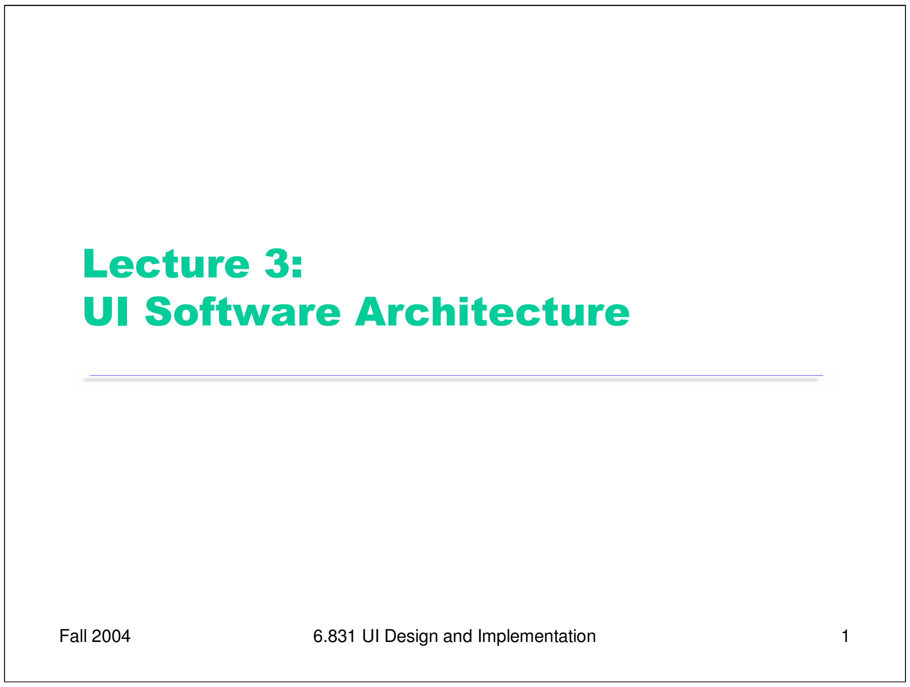
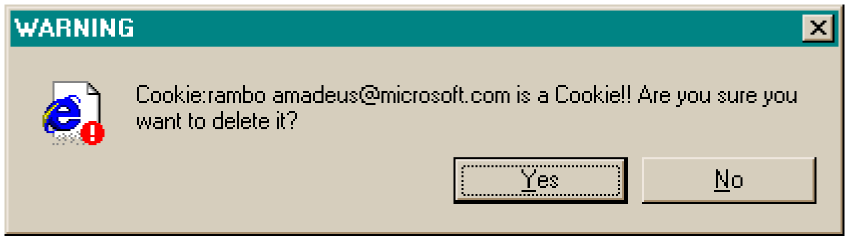


Lecture 3: UI Software Architecture

UI Hall of Fame or Shame?


WARNING

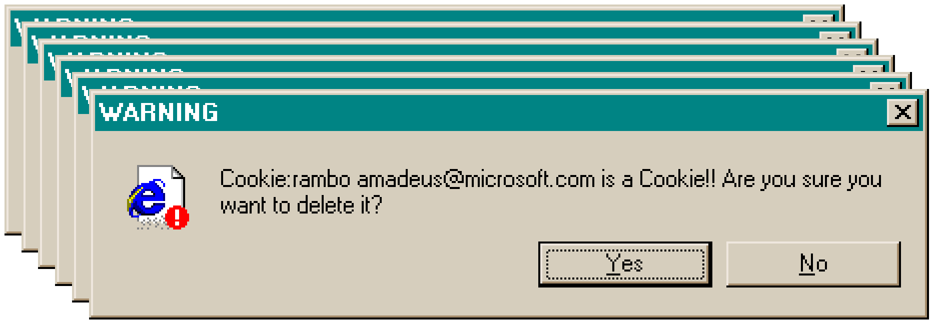
Cookie:rambo amadeus@microsoft.com is a Cookie!! Are you sure you want to delete it?

Source: Interface Hall of Shame

This message used to appear when you tried to delete the contents of your Internet Explorer cache from within the Windows Explorer.

Put aside the fact that the message is almost tautological (“Cookie… is a Cookie”) and overexcited (“!!”). Does it give the user enough information to make a decision? What’s a Cookie? What will happen if I delete it? Don’t ask questions the user can’t answer.

Hall of Shame


WARNING

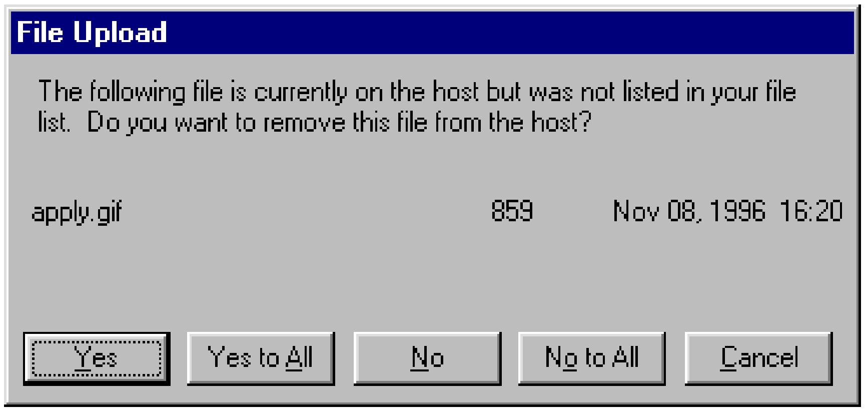
Cookie:rambo amadeus@microsoft.com is a Cookie!! Are you sure you want to delete it?

Source: Interface Hall of Shame

And definitely don’t ask more than once. There may be hundreds of cookies cached in the browser; this dialog box appears for each one the user selected.

There’s something missing from the dialog, whose absence becomes acute once the dialog appears a few times: a Cancel button. Always give users a way to **escape.**

Hall of Fame or Shame?


File Upload

The following file is currently on the host but was not listed in your file list. Do you want to remove this file from the host?

apply.gif 859 Nov 08, 1996 16:20

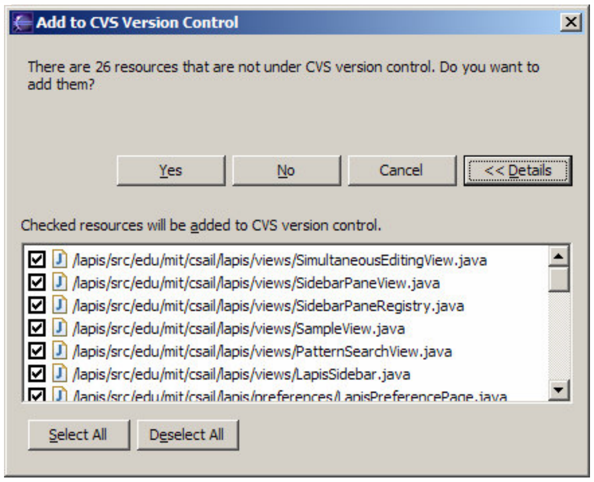
Source: Interface Hall of Shame

One way to fix the too-many-questions problem is Yes To All and No To All buttons, which short-circuit the rest of the questions by giving a blanket answer. That's a helpful **shortcut**, but this example shows that it's not a panacea.

This dialog is from Microsoft's Web Publishing Wizard, which uploads local files to a remote web site. Since the usual mode of operation in web publishing is to develop a complete copy of the web site locally, and then upload it to the web server all at once, the wizard suggests deleting files on the host that don't appear in the local files, since they may be orphans in the new version of the web site.

But what if you know there's a file on the host that you **don't** want to delete? You'd have to say No to every dialog until you found that file.

Hall of Fame



Add to CVS Version Control

There are 26 resources that are not under CVS version control. Do you want to add them?

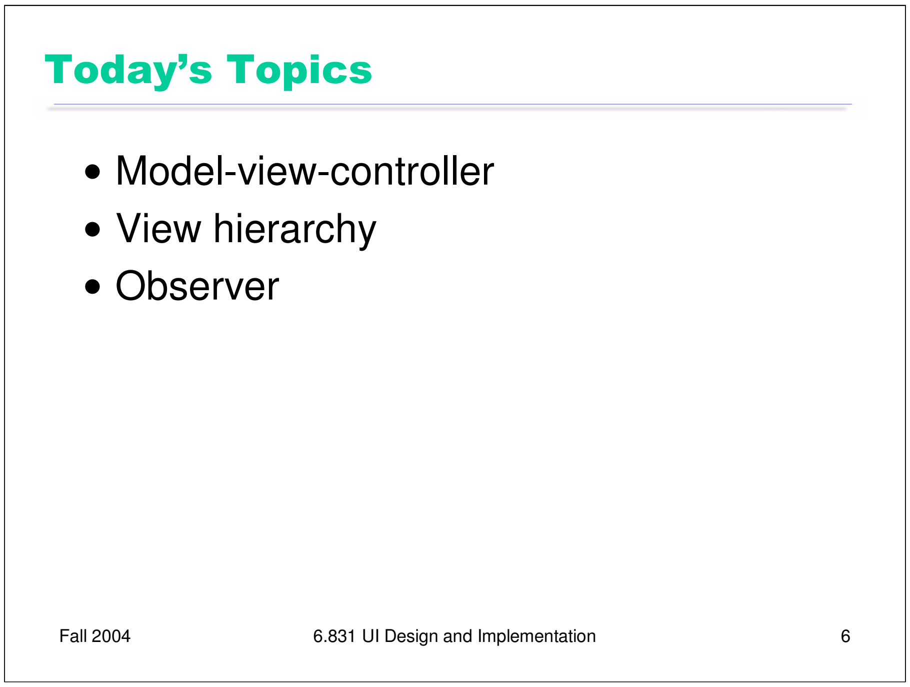
Checked resources will be added to CVS version control.

- ☑ /lapis/src/edu/mit/csail/lapis/views/SimultaneousEditingView.java
- ☑ /lapis/src/edu/mit/csail/lapis/views/SidebarPaneView.java
- ☑ /lapis/src/edu/mit/csail/lapis/views/SidebarPaneRegistry.java
- ☑ /lapis/src/edu/mit/csail/lapis/views/SampleView.java
- ☑ /lapis/src/edu/mit/csail/lapis/views/PatternSearchView.java
- ☑ /lapis/src/edu/mit/csail/lapis/views/LapisSidebar.java
- ☑ /lapis/src/edu/mit/csail/lapis/preferences/LapisPreferencePage.java

If your interface has a potentially large number of related questions to ask the user, it's much better to aggregate them into a single dialog. Provide a list of the files, and ask the user to select which ones should be deleted. Select All and Unselect All buttons would serve the role of Yes to All and No to All.

Here's an example of how to do it right, provided by IBM Eclipse. If there's anything to criticize in Eclipse's dialog box, it might be the fact that it initially doesn't show the filenames, just their count --- you have to press Details to see the whole dialog box. Simply knowing the *number* of files not under CVS control is rarely enough information to decide whether you want to say yes or no, so most users are likely to press Details anyway.

Nevertheless, this deserves to be in the hall of fame.

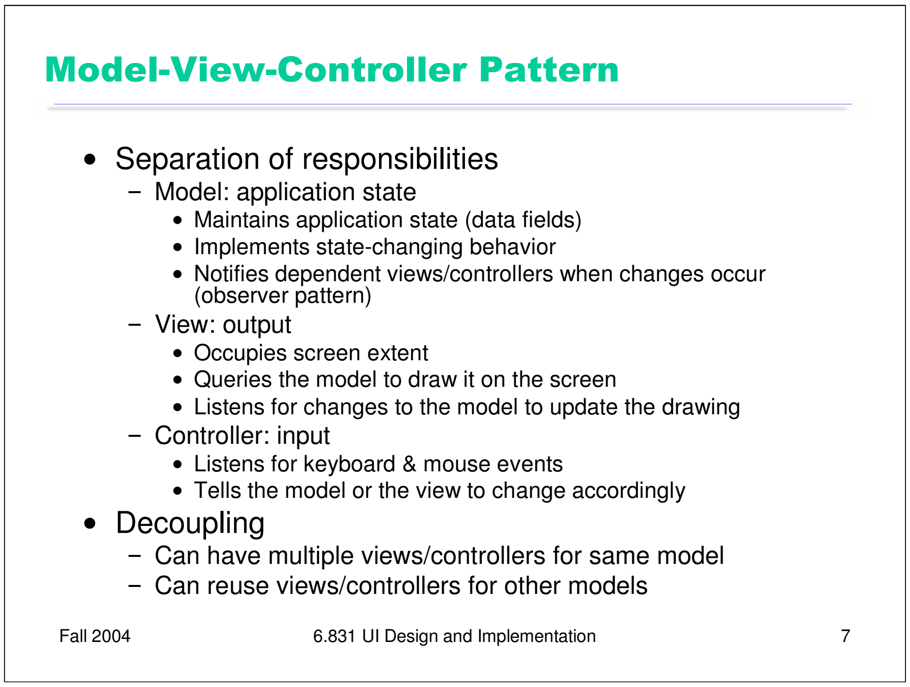


Today’s Topics

- Model-view-controller
- View hierarchy
- Observer

Fall 2004 6.831 UI Design and Implementation 6

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. Three of the most important patterns are the **model-view-controller** abstraction, which has evolved somewhat since its original formulation in the early 80’s; the **view hierarchy**, which is a central feature in the architecture of every popular GUI toolkit; and the **observer** pattern, which is essential to decoupling the model from the view and controller.



Model-View-Controller Pattern

- Separation of responsibilities
  - Model: application state
    - Maintains application state (data fields)
    - Implements state-changing behavior
    - Notifies dependent views/controllers when changes occur (observer pattern)
  - View: output
    - Occupies screen extent
    - Queries the model to draw it on the screen
    - Listens for changes to the model to update the drawing
  - Controller: input
    - Listens for keyboard & mouse events
    - Tells the model or the view to change accordingly
- Decoupling
  - Can have multiple views/controllers for same model
  - Can reuse views/controllers for other models

Fall 2004 6.831 UI Design and Implementation 7

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.

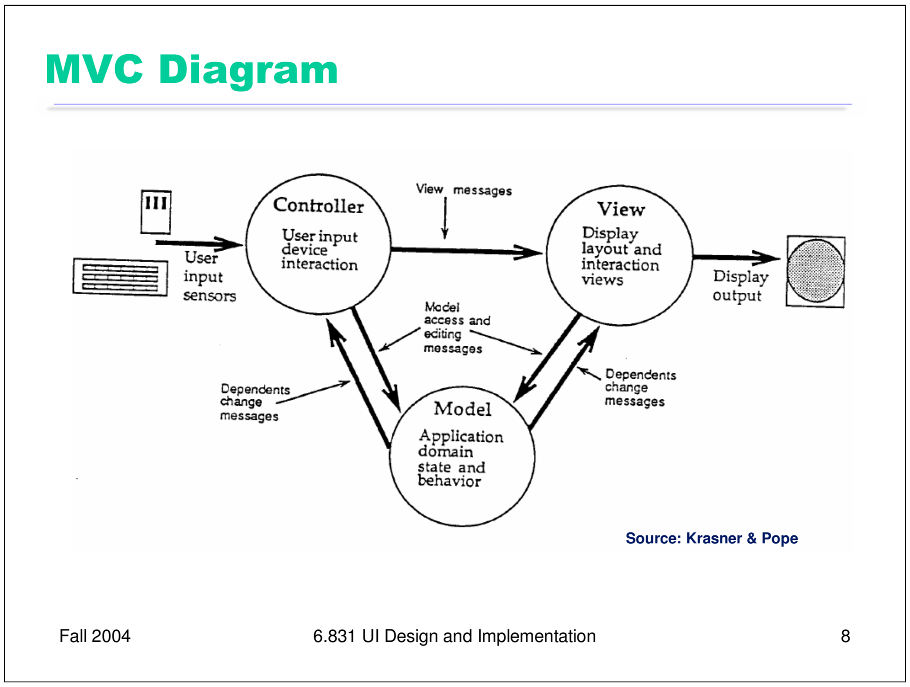
The model is responsible for maintaining application-specific data and providing access to that data. Models are often mutable, and they provide methods for changing the state safely, preserving its representation invariants. OK, all mutable objects do that. But a model must also notify its clients when 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.

View objects are responsible for output. A view usually 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.

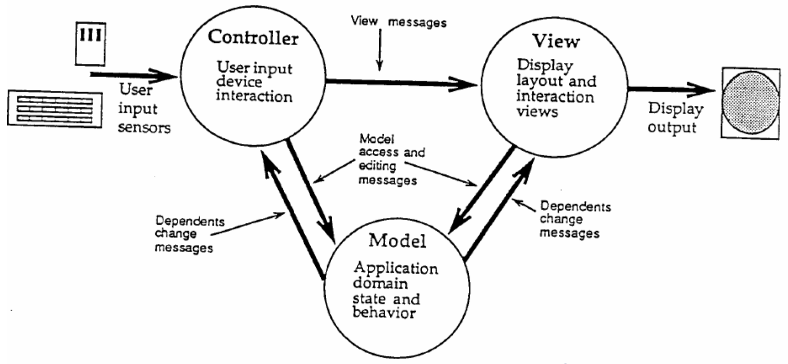
Finally, the controller handles the input. It receives keyboard and mouse events, and instructs the model to change accordingly.

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 is not really as well separated as we might like. We’ll see why a little later.



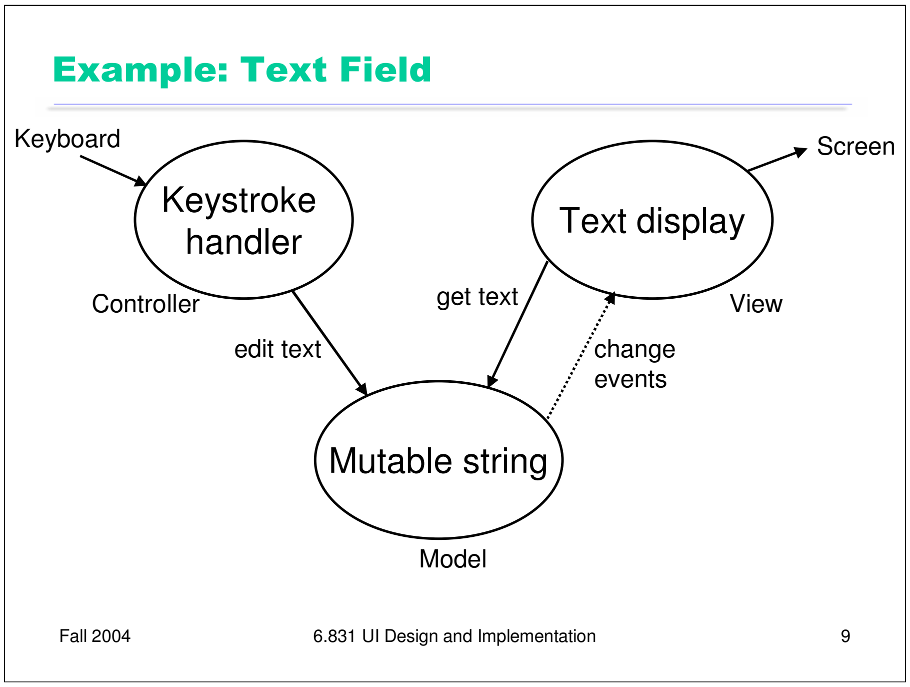
MVC Diagram


Controller
User input device interaction
View messages
View
Display layout and interaction views
Display output
User input sensors
Model access and editing messages
Dependents change messages
Model
Application domain state and behavior
Dependents change messages

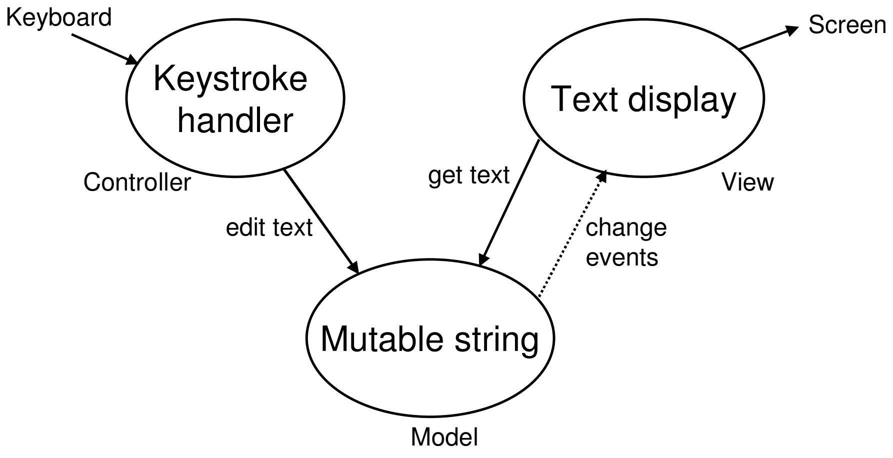
Source: Krasner & Pope

Fall 2004 6.831 UI Design and Implementation 8

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).

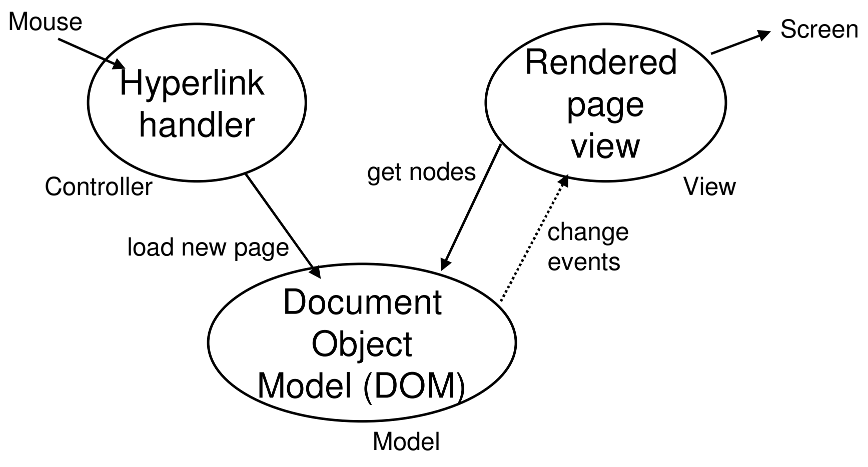


Example: Text Field



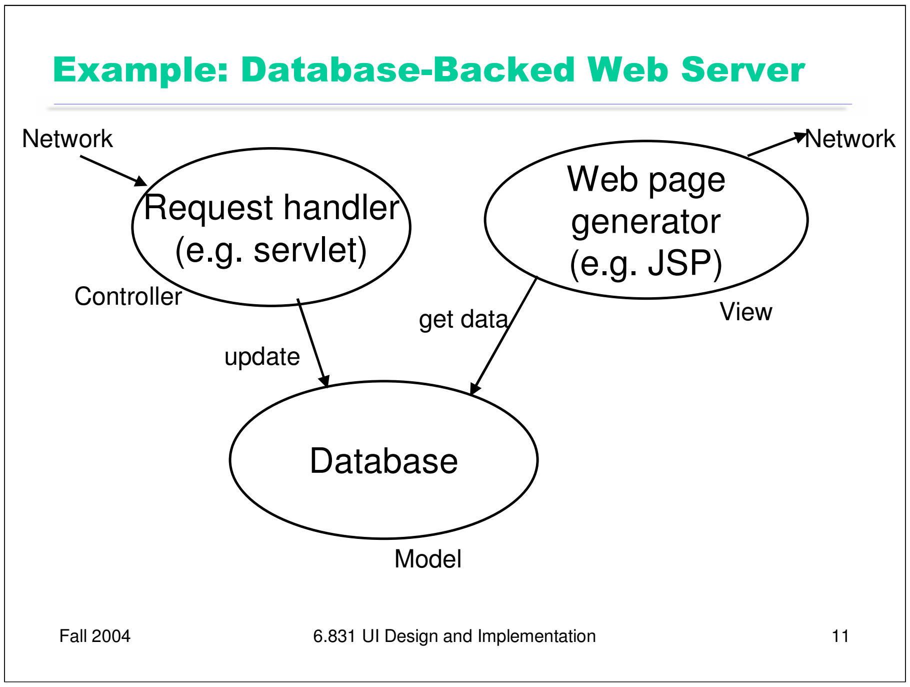
A simple example of the MVC pattern is a text field widget. Its model is a mutable string of characters. The view is an object that draws the text on the screen (usually with a rectangle around it to indicate that it's an editable text field). The controller is an object that receives keystrokes typed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

Example: Web Browser



Here's another example: a typical web browser. The model is a tree of nodes, called the Document Object Model, which represent the HTML elements of a web page. The view interprets these nodes to display the web page on the screen; for example, the contents of an H1 heading node would be displayed with a large, bold font. One controller is the hyperlink handler: when the user clicks on a hyperlink, the controller determines which URL the link points to, and then fetches a page from that URL and loads it into the model.

Web pages have lots of examples of MVC. A web page can contain its own controllers embedded in it, such as "onclick" attributes written in Javascript. Other attributes contain instructions for the view, such as the "style" attribute. And Javascript embedded in a web page can manipulate the model directly, by adding, removing, or modifying nodes in the Document Object Model.



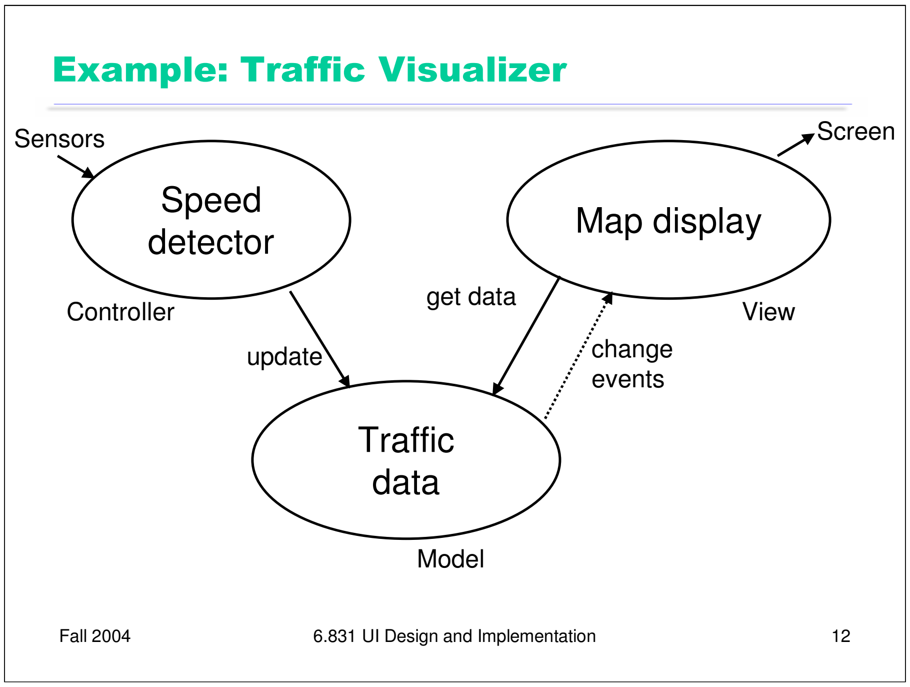
Example: Database-Backed Web Server

$$\begin{tikzcd}
\text{Network} \arrow[r] & \text{Request handler (e.g. servlet)} \arrow[d, "\text{update}"'] & & \text{Web page generator (e.g. JSP)} \arrow[r] \arrow[dll, "\text{get data}"'] & \text{Network} \\
& \text{Database} & & &
\end{tikzcd}$$

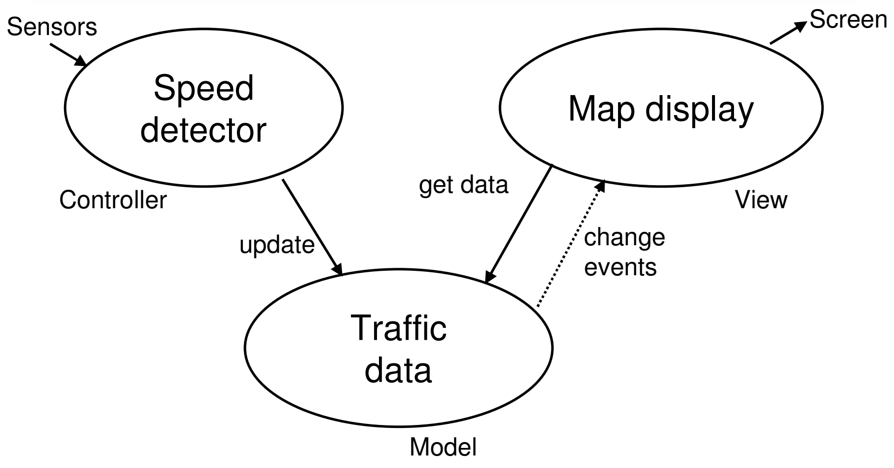
The MVC pattern can be seen in other kinds of user interfaces in which the input and output aren't strictly keyboard, mouse, and screen. Here's what a typical web server application looks like. The model is a database, usually a set of tables in a relational database like MySQL. The view is a template for a web page, typically described in a language like JSP, PHP, or ASP; when executed, this template requests data from the database in order to fill itself in. The output is a web page, which is delivered across a network connection; presumably there's a web browser on the other end to render it (though not necessarily!), but the web server isn't concerned with the actual screen display.

The user's inputs come back as a network request – e.g., a form filled out by the user. The controller interprets this request and uses it to update the database.

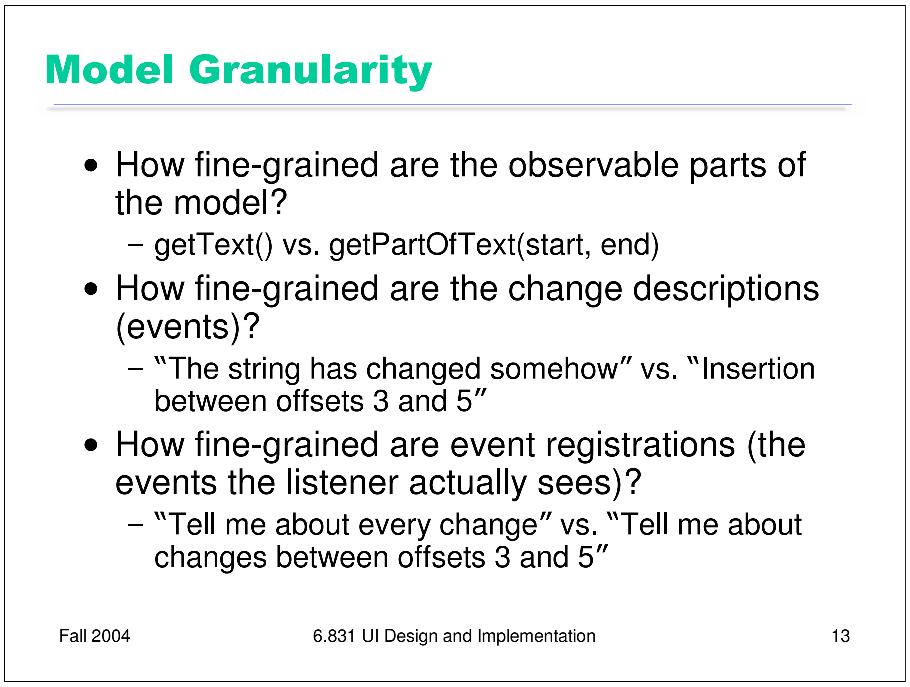
There are no mouse clicks or screen pixels in this example, but you can still use MVC to separate input, output, and backend. (But notice that the view isn't listening to the model in this example. Why not?)



Example: Traffic Visualizer



Here's one final example with a few more twists: a system that tracks traffic congestion for the entire United States. The model is again a database, which in this case stores a table of road segments with the current average traffic speed for each road segment. The view displays a map of the road segments, with congestion (slow traffic) highlighted. This view can be zoomed in to display only a small area that the user is interested in. Finally, the controller for the model takes input not from the user, but from a network of sensors distributed throughout the country's road network.



Model Granularity

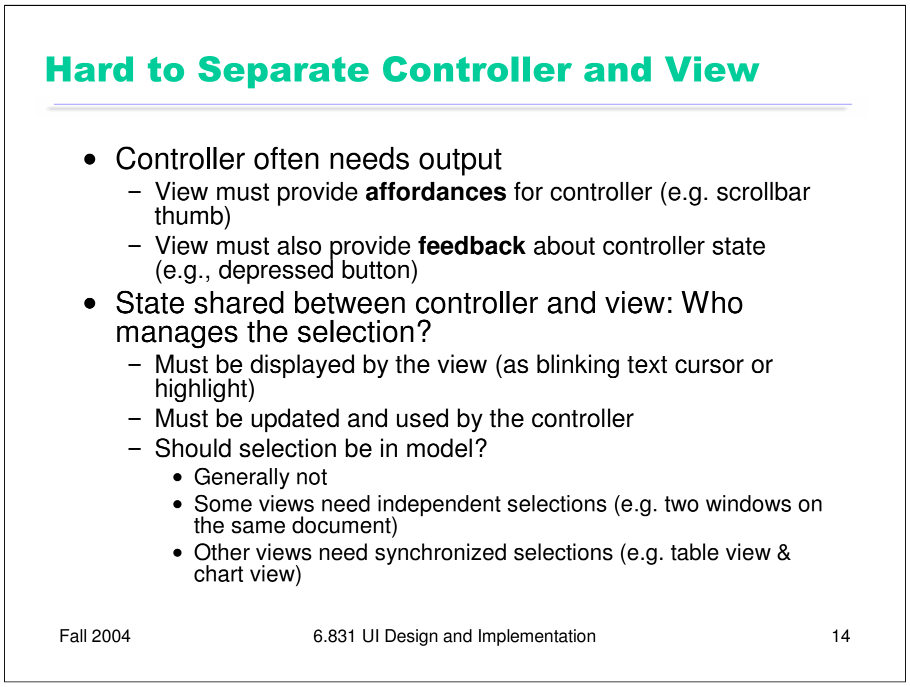
- How fine-grained are the observable parts of the model?
  - getText() vs. getPartOfText(start, end)
- How fine-grained are the change descriptions (events)?
  - "The string has changed somehow" vs. "Insertion between offsets 3 and 5"
- How fine-grained are event registrations (the events the listener actually sees)?
  - "Tell me about every change" vs. "Tell me about changes between offsets 3 and 5"

Fall 2004 6.831 UI Design and Implementation 13

Designing a model's 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. The traffic visualizer is worse – hundreds of thousands of streets, but only a few are actually important to the view if the map has been zoomed in. 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 to keep track of what changed.

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, like the traffic visualization, or like a long web page.



Hard to Separate Controller and View

- Controller often needs output
  - View must provide **affordances** for controller (e.g. scrollbar thumb)
  - View must also provide **feedback** about controller state (e.g., depressed button)
- State shared between controller and view: Who manages the selection?
  - Must be displayed by the view (as blinking text cursor or highlight)
  - Must be updated and used by the controller
  - 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)

Fall 2004 6.831 UI Design and Implementation 14

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.

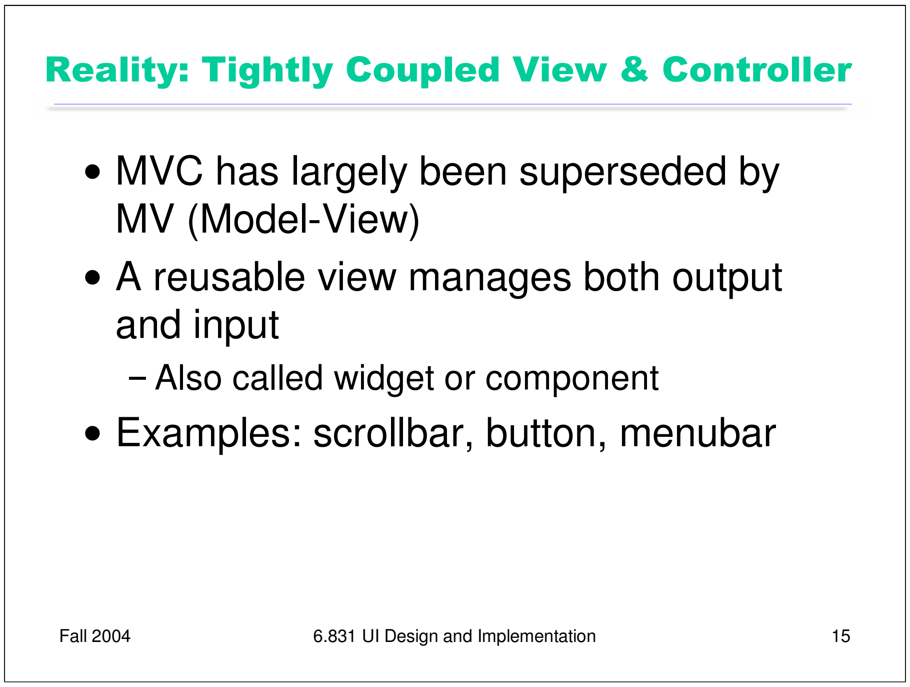
First, a controller often needs to produce its own output. The view must display **affordances** for the controller, such as selection handles or scrollbar thumbs. The controller must be aware of the screen locations of these affordances. When the user starts manipulating, the view must modify its appearance to give **feedback** about the manipulation, e.g. painting a button as if it were depressed.

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.



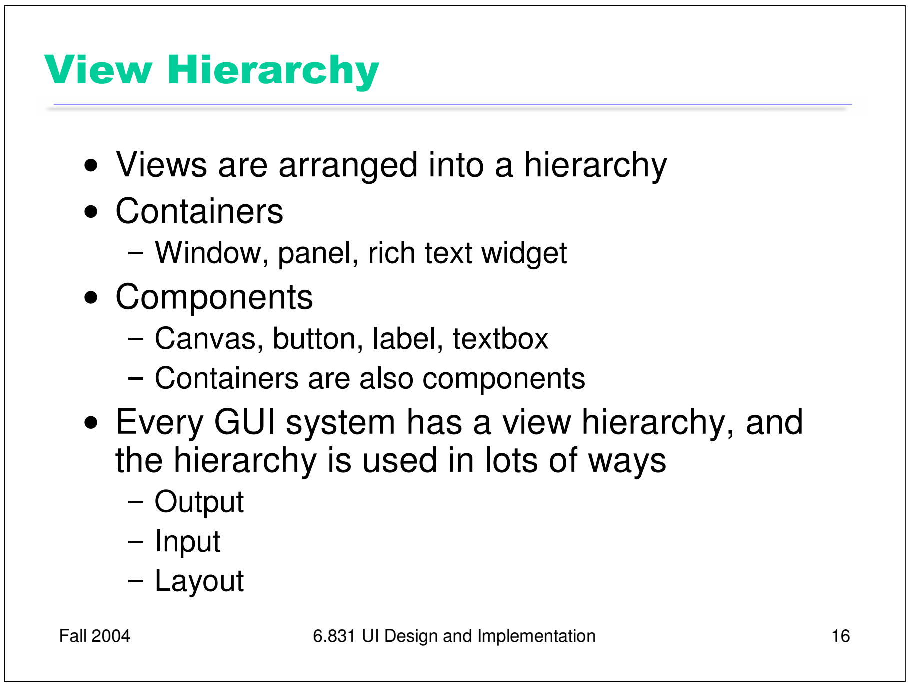
Reality: Tightly Coupled View & Controller

- MVC has largely been superseded by MV (Model-View)
- A reusable view manages both output and input
  - Also called widget or component
- Examples: scrollbar, button, menubar

Fall 2004 6.831 UI Design and Implementation 15

In principle, it's a nice idea to separate input and output into separate, reusable classes. In reality, it isn't always feasible, because input and output are tightly coupled in graphical user interfaces. As a result, the MVC pattern has largely been superseded by what might be called Model-View, in which the view and controllers are fused together into a single class, often called a **component** or a **widget**.

Most of the widgets in the Swing library are fused view/controllers like this; you can't, for example, pull out JScrollbar's controller and reuse it in your own custom scrollbar. Internally, JScrollbar follows a model-view-controller architecture, but the view and controller aren't independently reusable.



View Hierarchy

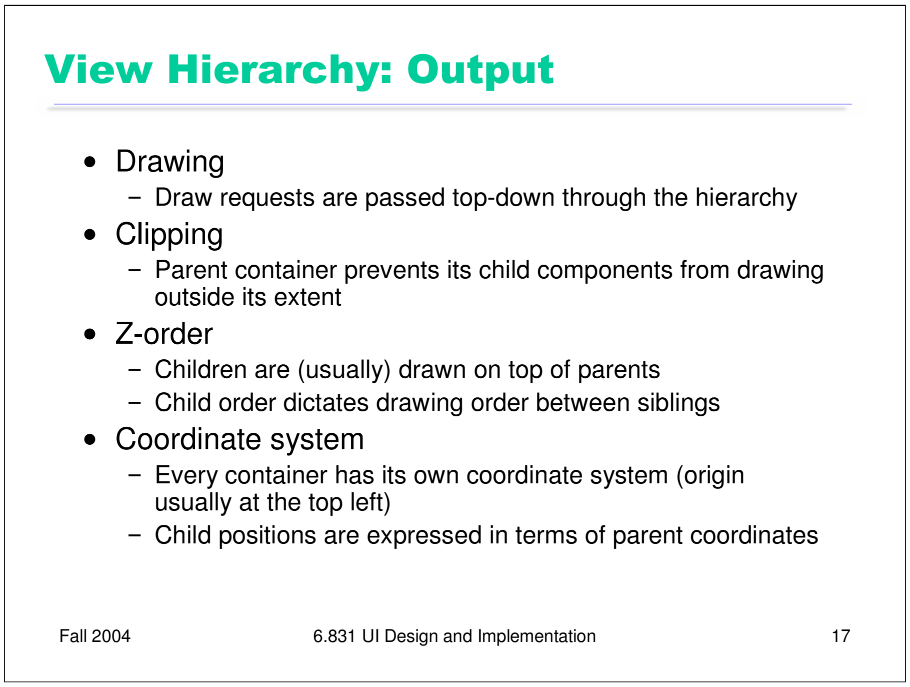
- Views are arranged into a hierarchy
- Containers
  - Window, panel, rich text widget
- Components
  - Canvas, button, label, textbox
  - Containers are also components
- Every GUI system has a view hierarchy, and the hierarchy is used in lots of ways
  - Output
  - Input
  - Layout

Fall 2004 6.831 UI Design and Implementation 16

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

Views are arranged into a hierarchy of containment, in 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

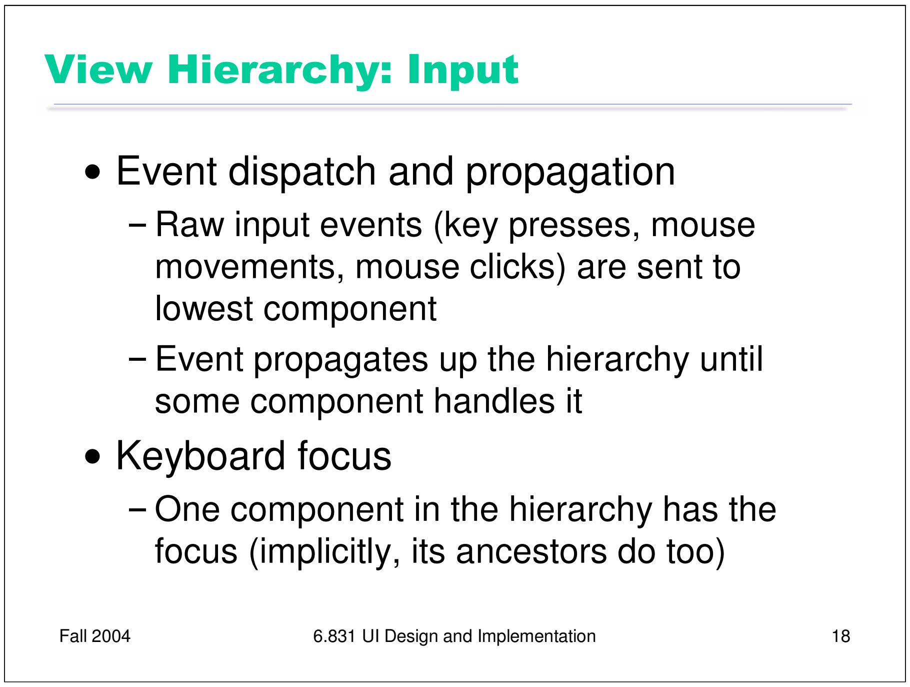
Fall 2004 6.831 UI Design and Implementation 17

First, and probably most important, 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

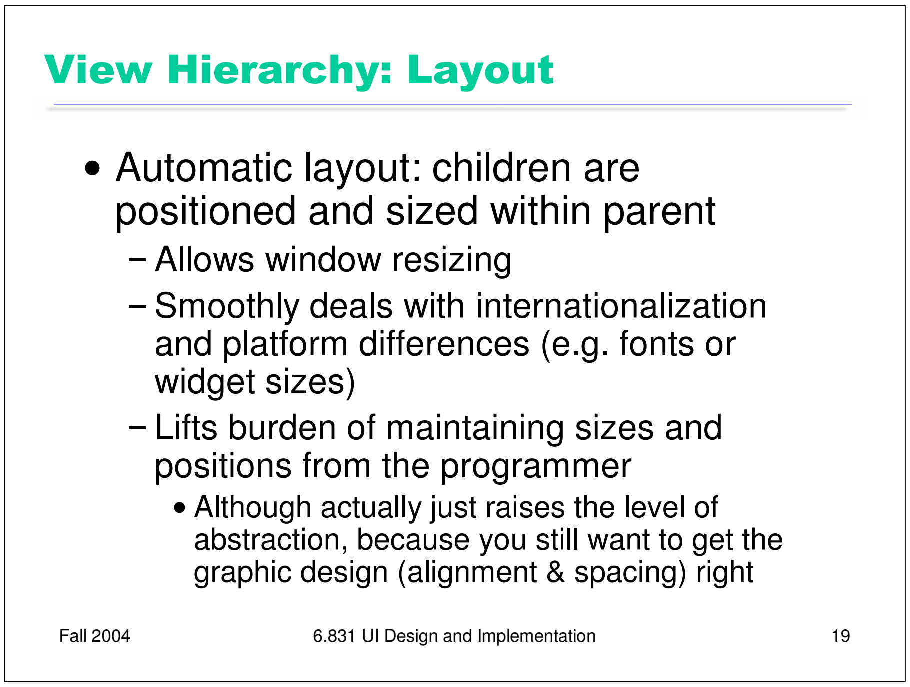
- Event dispatch and propagation
  - Raw input events (key presses, mouse movements, mouse clicks) are sent to lowest component
  - Event propagates up the hierarchy until some component handles it
- Keyboard focus
  - One component in the hierarchy has the focus (implicitly, its ancestors do too)

Fall 2004 6.831 UI Design and Implementation 18

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.



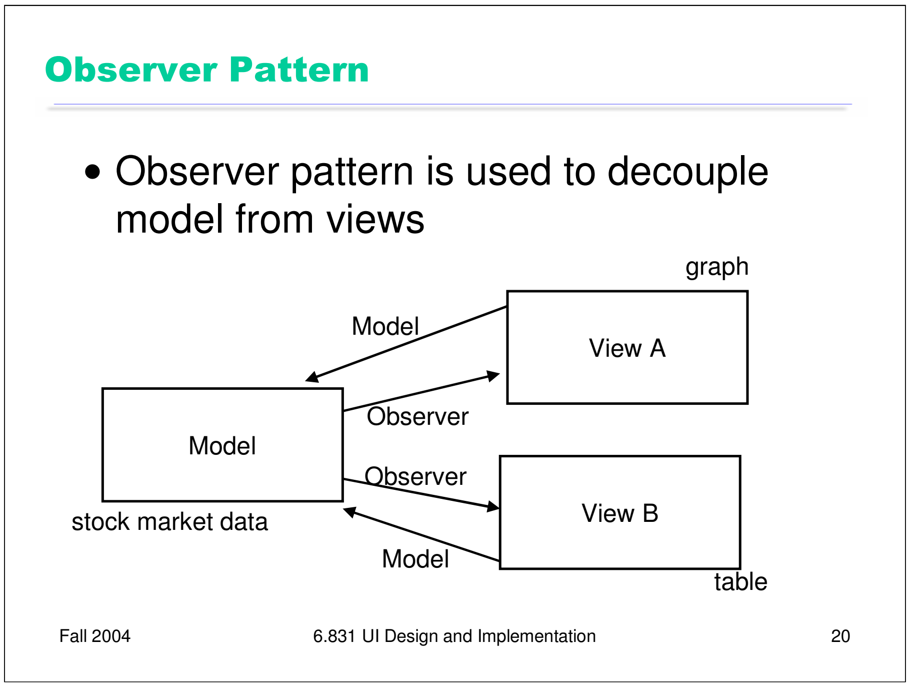
View Hierarchy: Layout

- 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

Fall 2004 6.831 UI Design and Implementation 19

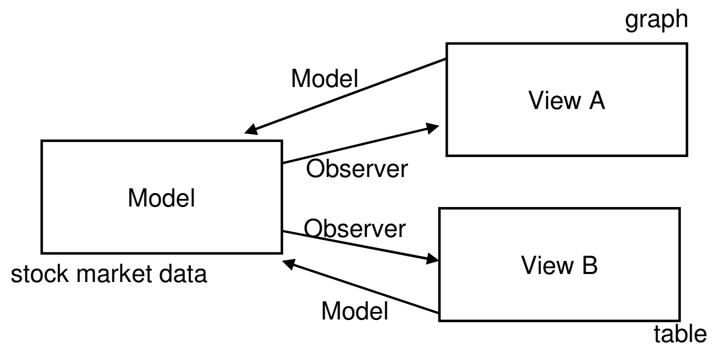
The view hierarchy is also used to direct the **layout** process, which determines the extents (positions and sizes) of the views in the hierarchy. Many GUI systems have supported automatic layout, including Motif (an important early toolkit for X Windows), Tk (a toolkit developed for the Tcl scripting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

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.



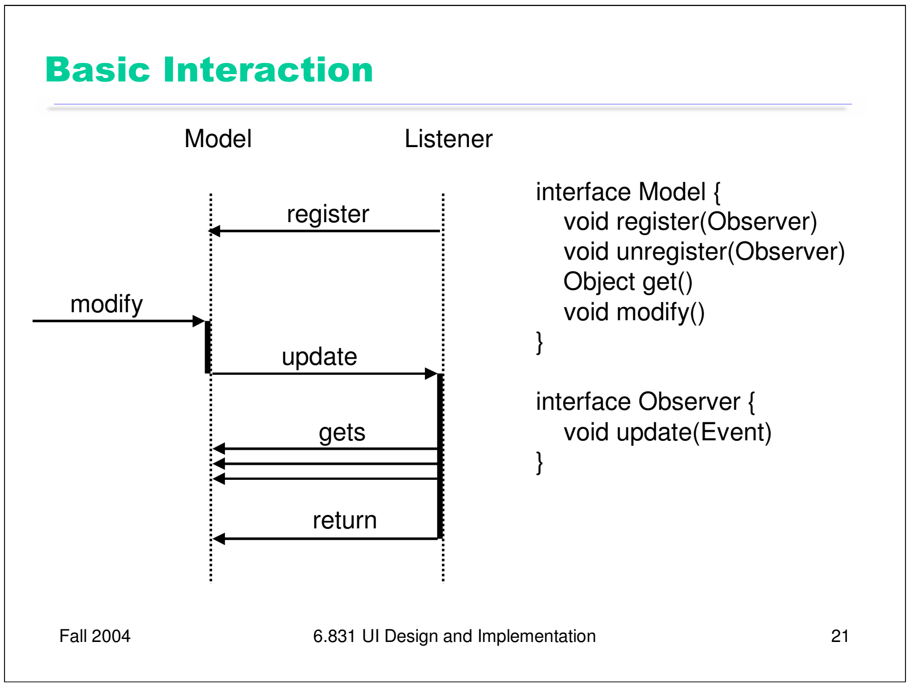
Observer Pattern

- Observer pattern is used to decouple model from views

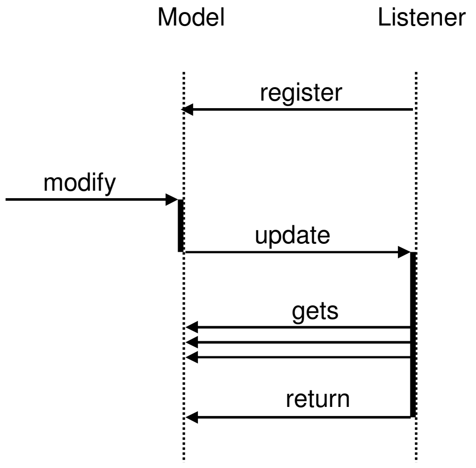


Fall 2004 6.831 UI Design and Implementation 20

Finally, let’s look at the **observer pattern**. Observers are called listeners in Java; we’ll use the two terms interchangeably in this course.



Basic Interaction



```
interface Model {
    void register(Observer)
    void unregister(Observer)
    Object get()
    void modify()
}

interface Observer {
    void update(Event)
}
```

Fall 2004 6.831 UI Design and Implementation 21

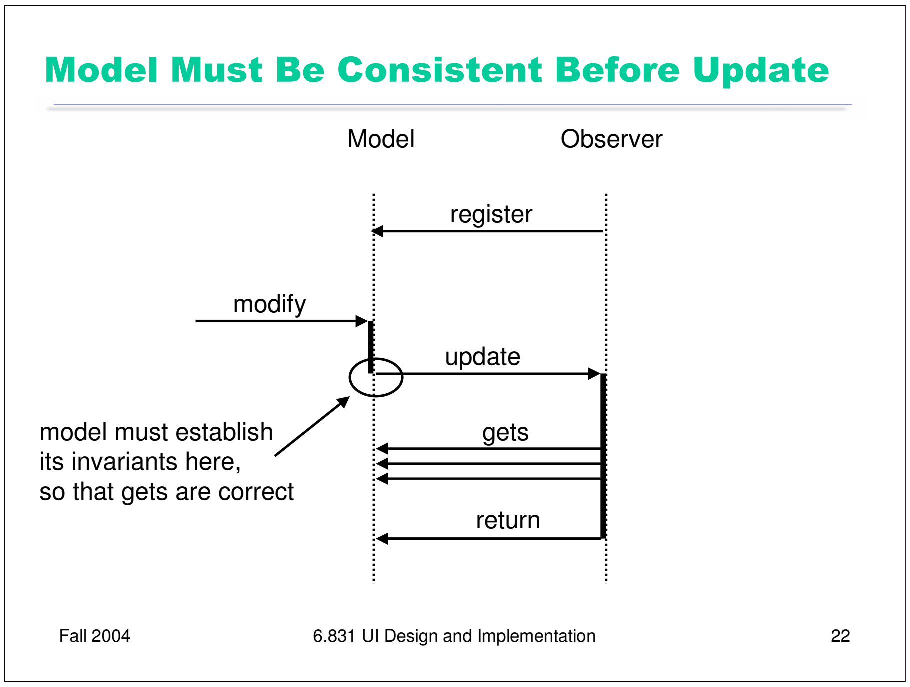
Here's the conventional interaction that occurs in the observer pattern. (We'll use the abstract representation of Model and Observer shown on the right. Real models and observers will have different, more specific names for the methods, and different method signatures. They'll also have multiple versions of each of these methods.)

1. An observer **registers** itself to receive notifications from the model.

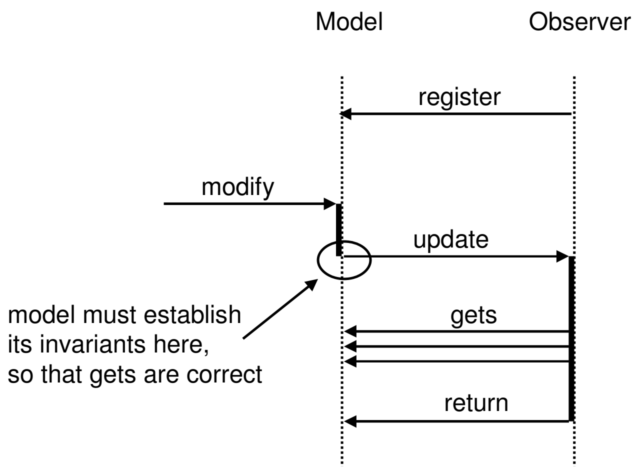
2. When the model changes (usually due to some other object **modifying** it), the model broadcasts the change to all its registered views by calling **update** on them. The update call usually includes some information about what change occurred. One way is to have different update methods on the observer for each kind of change (e.g. treeStructureAdded() vs. treeStructureRemoved()). Another way is to package the change information into an **event** object. Regardless of how it's packaged, this change information that is volunteered by the model is usually called **pushed** data.

3. An observer reacts to the change in the model, often by **pulling** other data from the model using **get** calls.

We already discussed the tradeoff between fine-grained and coarse-grained registration and notification. There's also a tradeoff between pushing and pulling data.



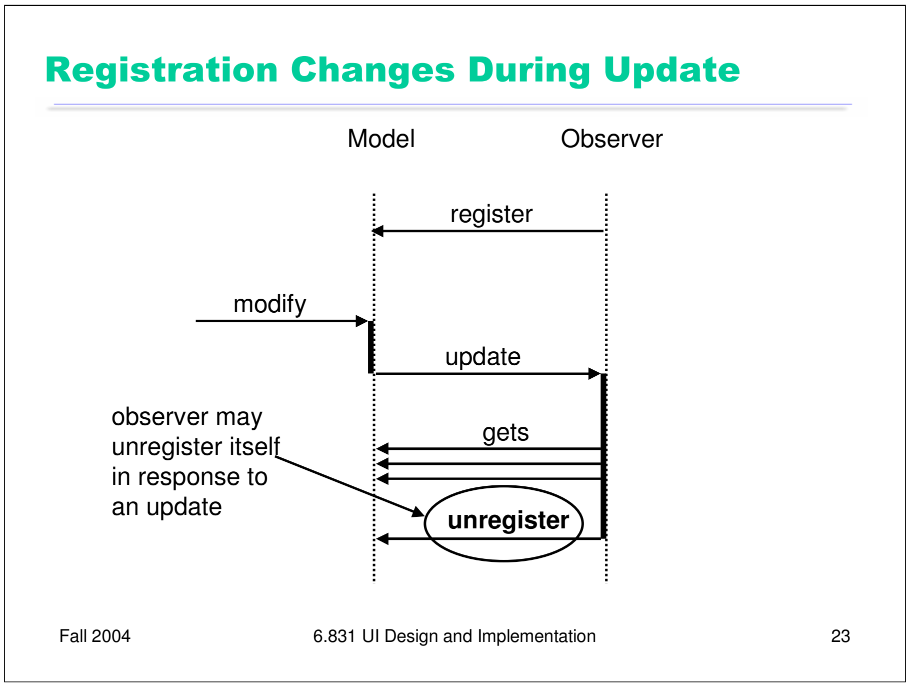
Model Must Be Consistent Before Update


Model
Observer
register
modify
update
model must establish its invariants here, so that gets are correct
gets
return

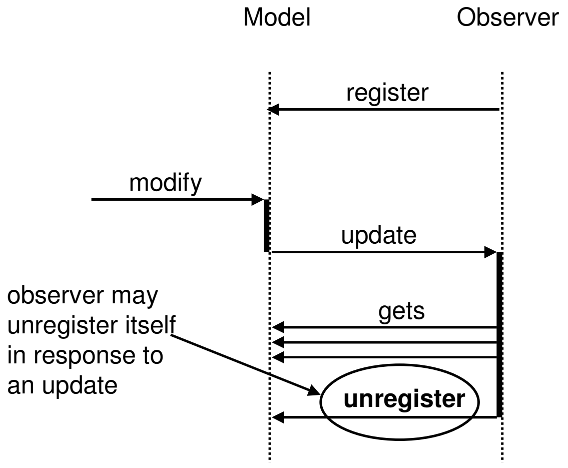
Fall 2004 6.831 UI Design and Implementation 22

Let’s talk about some important issues. First, when the model calls **update** on its observers, it is giving up control – in much the same way that a method gives up control when it returns to its caller. Observers are free to call back into the model, and in fact often do in order to pull information from it. So the model has to make sure that it’s consistent --- i.e., that it has established all of its internal invariants – before it starts issuing notifications to observers.

So it’s often best to delay firing off your observers until the end of the method that caused the modification. Don’t fire observers while you’re in the midst of making changes to the model’s data structure.



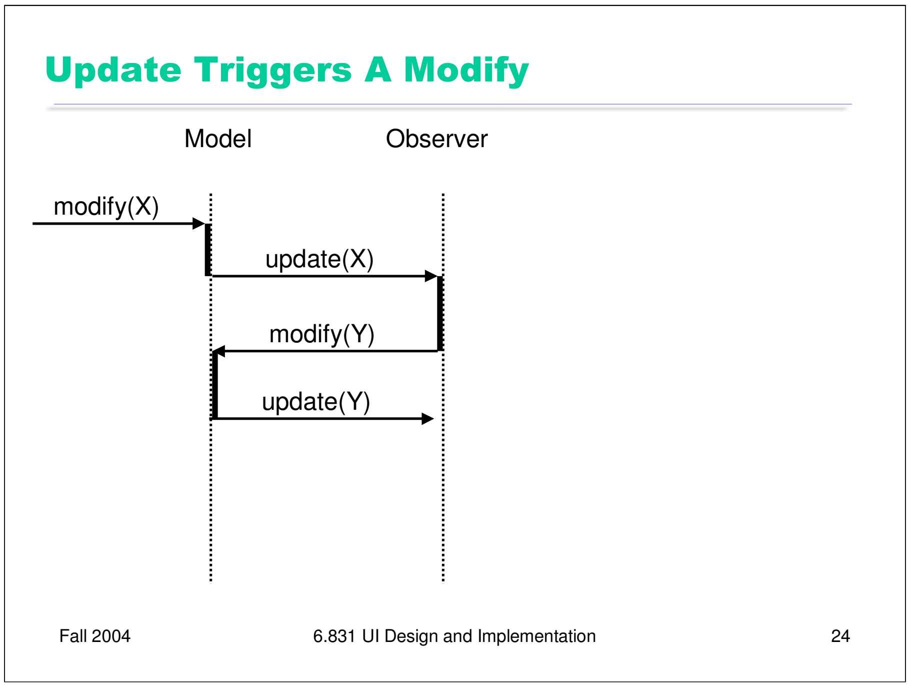
Registration Changes During Update



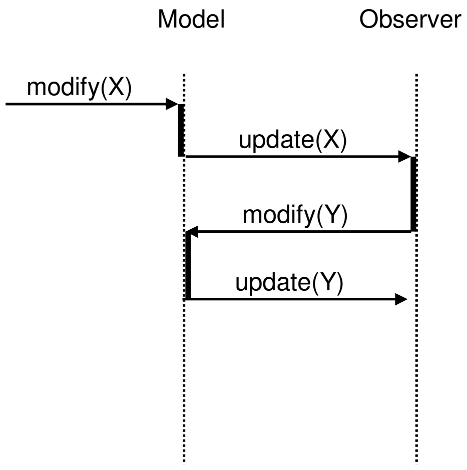
Fall 2004 6.831 UI Design and Implementation 23

Another potential pitfall is observers that unregister themselves. For example, suppose the model contains stock market data, and a view registers itself as an observer of one stock in order to watch for that stock reaches a certain price. Once the stock hits the target price, the view does its thing (e.g., popping up a window to notify the user); but then it's no longer needed, so it unregisters itself from the model.

This is a problem if the model is iterating naively over its collection of observers, and the collection is allowed to change in the midst of the iteration. It's safer to iterate over a *copy* of the observer list. Since one-shot observers are not particularly common, however, this imposes an extra cost on every event broadcast. So the ideal solution is to copy the observer list only when necessary – i.e., when a register or unregister occurs in the midst of event dispatch.

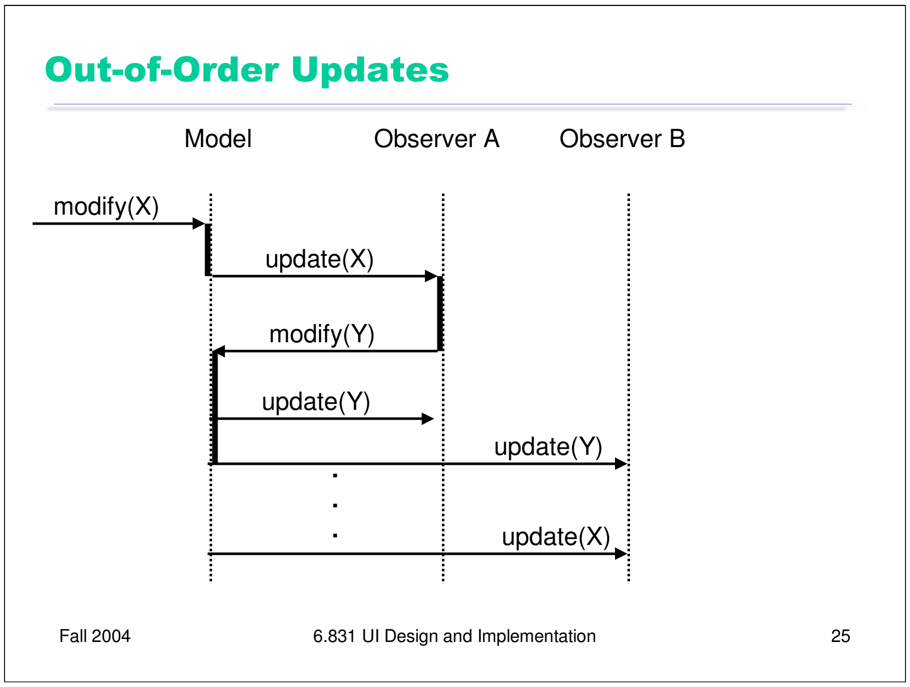


Update Triggers A Modify



Fall 2004 6.831 UI Design and Implementation 24

A third pitfall occurs when an observer responds to an update message by calling **modify** on the model. Why would it do that? It might, for instance, be trying to keep the model within some legal range. Obviously, this could lead to infinite regress if you're not careful. A good practice for models to protect themselves against sloppy views is to only send updates if a change actually occurs; if a client calls modify() but it has no actual effect on the model, then no updates should be sent.



Out-of-Order Updates

Fall 2004 6.831 UI Design and Implementation 25

A more pernicious pitfall can arise when there are multiple observers and one of them modifies the model: events can get out of order. This diagram shows an example of what can happen. Observer A gets the first update (for change X), and it responds by modifying the model. The model in turn responds by sending out another round of updates (change Y) immediately. If observer A makes no further modifications, observer B finally gets its updates – but it gets the changes in the opposite order that they actually occurred, change Y before change X!

There are a few solutions to this problem:

- the model could delay broadcasting event Y until all the updates for X have been sent. This is usually done by putting the events on a queue. It imposes some additional cost and complexity on the model, but it's the best way to guarantee that events arrive in the same order to all observers.

- the model could skip sending the update(X) to observer B. This ensures that observer B doesn't get an event with old data in it, but it also means that B has missed a transition. Some observers might care about those transitions: for example, if B is a graph displaying stock prices over a time interval.

- observers could ignore pushed data (X, Y) and always get the latest state directly from the model. This is good practice in general. If your view only needs to show the current model state, then get it directly from the model; don't rely on the pushed event to tell you what it is.