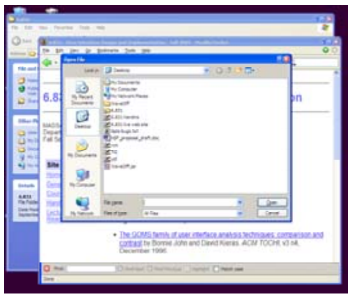
Lecture 3: UI Software Architecture

UI Hall of Fame or Shame?



Suggested by Vikki Chou

Today's candidate for the Hall of Fame or Shame is the **modal dialog box**.

A modal dialog box (like the File Open dialog seen here) prevents the user from interacting with the application that popped it up.

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).

But there are usability disadvantages too, chief among them **loss of user control** and reduced **visibility** (e.g., you can't see important information or previews in the main window, and can't scroll the main window to bring something else into view). Modal dialogs may also overload the user's **short-term memory** – if the user needs some information from the main window, or worse, from a second modal dialog, then they're forced to remember it, rather than simply viewing and interacting with both dialogs side-by-side.

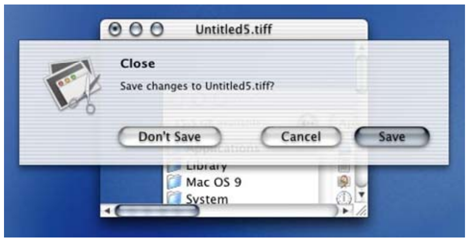
When you try to interact with the main window, Windows gives some nice animated **feedback** – flashing the border of the modal dialog box. This helps explain why your clicks on the main window had no effect.

On most platforms, you can at least move, resize, and minimize the main window, even when a modal dialog is showing. (The modal dialog minimizes along with it.) Alas, not on Windows…

the main window is completely pinned! You can minimize it only by obscure means, like the Show Desktop command, which minimizes *all* windows. This is a big obstacle to user control and freedom.

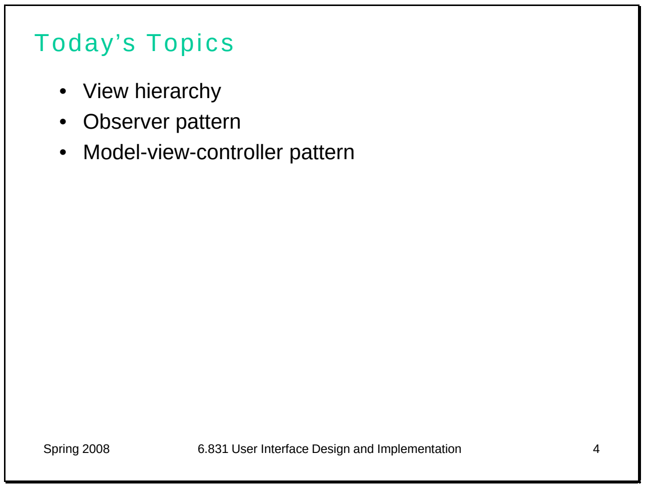
Modeless dialogs, by contrast, don't prevent using other windows in the application. They're often used for ongoing interactions with the main window, like Find/Replace. One problem is that a modeless dialog box can get in the way of viewing or interacting with the main window (as when a Find/Replace dialog covers up the match). Another problem is a **consistency** problem: modal dialogs and modeless dialogs usually look identical. Sometimes the presence of a Minimize button is a clue that it's modeless, but that's not a very strong visual distinction. 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.

UI Hall of Fame or Shame?



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.

Another advantage of Mac sheets is that they make a strong contrast with modeless dialogs – the translucent, anchored modal sheet is easy to distinguish from a modeless window.

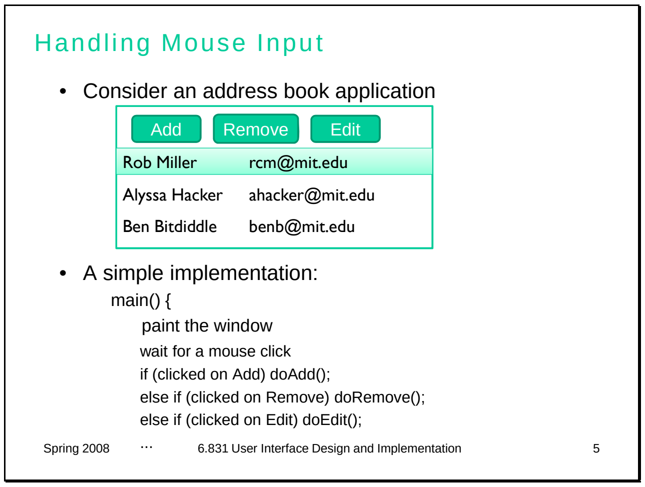


Today's Topics

- View hierarchy
- Observer pattern
- Model-view-controller pattern

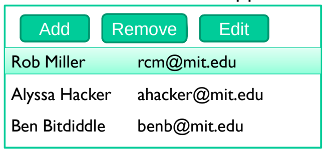
Spring 2008 6.831 User Interface Design and Implementation 4

Today's lecture is the first in the stream of lectures 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 important GUI toolkit; and the **observer** pattern, which is essential to decoupling the model from the view and controller.



Handling Mouse Input

- Consider an address book application



Rob Miller	rcm@mit.edu
Alyssa Hacker	ahacker@mit.edu
Ben Bitdiddle	benb@mit.edu

- A simple implementation:

```
main() {
    paint the window
    wait for a mouse click
    if (clicked on Add) doAdd();
    else if (clicked on Remove) doRemove();
    else if (clicked on Edit) doEdit();
    …
```

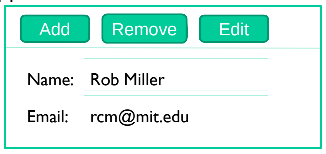
Spring 2008 6.831 User Interface Design and Implementation 5

To motivate these patterns, let's start with a simple example: how we might handle mouse input to a simple user interface. This program has a couple of buttons (Add, Remove, Edit). A simple way to do it might treat the mouse input as if it were a stream of commands, and have a simple loop that reads a mouse click, checks to see which button the click refers to (which you'd do by comparing the mouse pointer position against the button's rectangle), and then invokes a function.

This is in fact how many old menu-driven programs were implemented. The program prints a menu (or just a bare prompt). Then it sits and waits for you to enter a choice. Then it dispatches on that choice. At every point in the program, the set of things you can do (the set of inputs that the program expects) is hardcoded into the program, so the **system controls the dialog**.

Who Controls the Dialog: the User or the System?

- Suppose the user clicks Edit...



Name: Rob Miller

Email: rcm@mit.edu

```
void doEdit() {
   change window to show Name and Email textboxes
   put cursor in Name textbox
   wait for keyboard entry into Name textbox
   put cursor in Email textbox
   wait for keyboard entry into Email textbox
}
```

That simple approach, in which the system runs through a hardcoded dialog, doesn't work for graphical user interfaces. We don't want the system to be in control of the dialog – the **user should have the freedom** to decide what they need to interact with next. I shouldn't have to wait until the system asks me to enter something in the Email textbox; I should be able to interact with anything that's visible, at any time. But that wreaks havoc with a hardcoded dialog design. Not only should doEdit() be able to handle the Name and Email textboxes in any order, but it should be able to handle clicks on the Add and Remove buttons as well! As interfaces get bigger, hardcoded input handling simply can't provide the kind of freedom we need.

(As we saw in today's hall of fame & shame, of course, GUIs do still have modal dialogs in which the system restricts your choices. But they should be used sparingly, not for everything!)

Decouple Input Handling

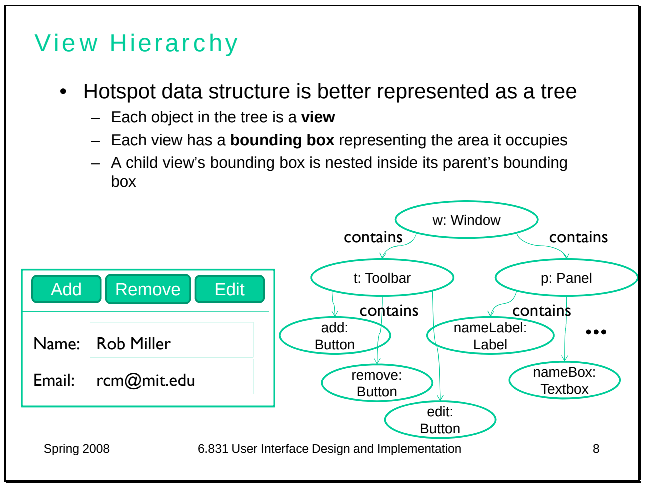
- Key idea: represent the input-handling code as a data structure

```
hotspots = (( Add , doAdd),
            ( Remove , doRemove),
            ( Rob Miller , selectName), ...)
```

  – Now the input handling code might look like this:

```
read mouse click
for each hotspot ∊ hotspots {
   if (clicked in hotspot.rectangle) hotspot.handler()
}
```

Since we can't hardcode the input handling to a particular point in the program, we represent the set of possible inputs with a data structure instead. Here's the simplest data structure we might imagine: a list of **hotspots** on the screen where a mouse click causes the program to do something. Some primitive UI toolkits do use this approach; the HTML imagemap element is a good example.



View Hierarchy

- Hotspot data structure is better represented as a tree
  - Each object in the tree is a **view**
  - Each view has a **bounding box** representing the area it occupies
  - A child view's bounding box is nested inside its parent's bounding box

Add
Remove
Edit
Name: Rob Miller
Email: rcm@mit.edu
w: Window
contains
contains
t: Toolbar
p: Panel
contains
contains
add: Button
remove: Button
edit: Button
nameLabel: Label
nameBox: Textbox
•••

Spring 2008 6.831 User Interface Design and Implementation 8

In general, however, it works better to structure the hotspots as a tree, so that the user interface can be implemented in a modular way.

This leads to the first important pattern we'll talk about today: the **view hierarchy.** A view is an object that covers a certain area of the screen, generally a rectangular area called its bounding box. The view concept goes by a variety of names in various UI toolkits. In Java Swing, they're JComponents; in HTML, they're elements; in other toolkits, they may be called widgets, controls, or interactors.

Views are arranged into a hierarchy of containment, in which some views (called containers in the Java nomenclature) can contain other views. Typical containers are windows, panels, and toolbars. The view hierarchy is not just an arbitrary tree, but is in fact a spatial hierarchy: child views are nested inside their parent's bounding box.

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:

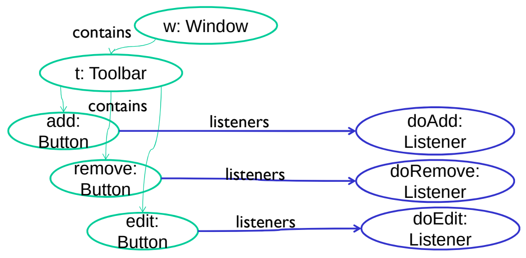
Output. Views are responsible for displaying themselves, and the view hierarchy directs the display process.

Input. Views can have input handlers, and the view hierarchy controls how mouse and keyboard input is processed.

Layout. The view hierarchy controls how the views are laid out on the screen, i.e. how their bounding boxes are assigned.

Input Handling

- Input handlers are associated with views



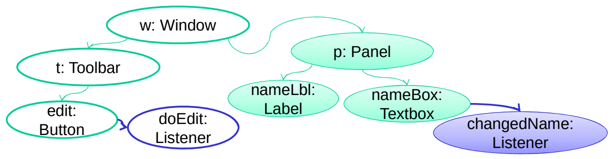
Spring 2008 6.831 User Interface Design and Implementation 9

To handle mouse input, for example, each view can act as a hotspot, and we can attach a handler to the view that is called when the mouse is clicked on it. Thus the view hierarchy subsumes our simple hotspot-list data structure.

Handlers are also called **listeners**, event handlers, subscribers, and observers.

Event-Based Programming

- Control flow through a graphical user interface
  - Top-level loop (**event loop**) reads all input from mouse and keyboard
  - Listener changes state of the interface (e.g. modifying the view hierarchy) and returns immediately to the event loop



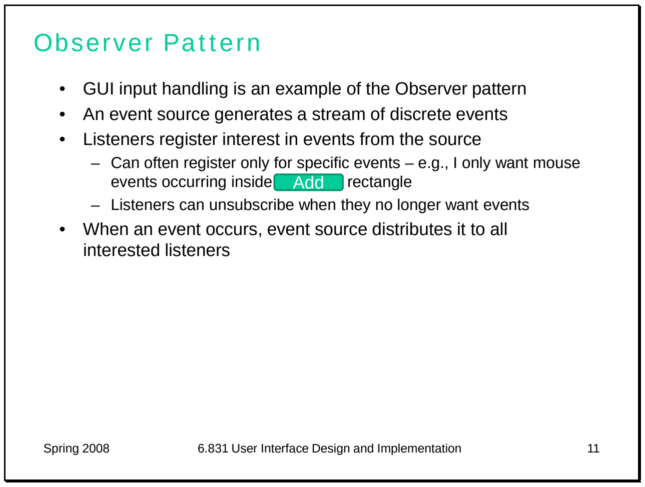
Spring 2008 6.831 User Interface Design and Implementation 10

This idea – structuring a user interface as a output view hierarchy with input listeners attached to the views – gives rise to the essential paradigm of GUI programming, in which everything happens in response to **events**.

The top level of a GUI program is an **event loop** which is responsible for reading from the mouse and keyboard. In many GUI toolkits, this loop is actually invisible, buried inside the toolkit runtime system; you don't write it yourself. (Java Swing is like this; so is HTML and Javascript.)

For each input event, the event loop finds the appropriate view in the view hierarchy (for example, by looking at the x,y position of the mouse when a mouse click occurred) and calls the listener(s) attached to it. The listeners react by changing the state of the interface, but then return immediately to the event loop. For example, doEdit() (the listener for the Edit button) might create the Name and Email textboxes and attach them to the view hierarchy, but it **doesn't wait** for the user to actually enter anything in the textboxes. Instead, additional listeners are attached to the textboxes to take care of that later, and the event loop does the waiting.

This is a simplification of the process – we'll dig into how input is actually processed in a future lecture. But it's important to understand that there is no straight-line control flow through an event-based program, starting from the main() function and passing through the rest of the program in a predictable way. Instead, primary control is held by the event loop, and doled out in little sips to input event handlers.

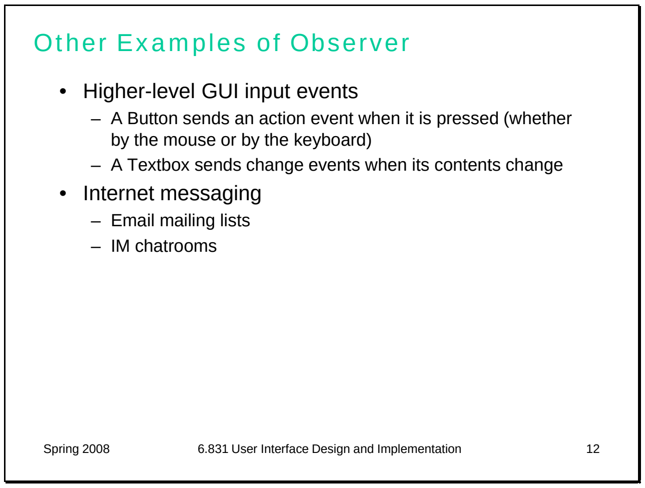

Observer Pattern

- GUI input handling is an example of the Observer pattern
- An event source generates a stream of discrete events
- Listeners register interest in events from the source
  - Can often register only for specific events – e.g., I only want mouse events occurring inside Add rectangle
  - Listeners can unsubscribe when they no longer want events
- When an event occurs, event source distributes it to all interested listeners

Spring 2008 6.831 User Interface Design and Implementation 11

GUI input event handling is an instance of the Observer pattern (also known as Listener and Publish-Subscribe). In the Observer pattern, an event source generates a stream of discrete events, which correspond to state transitions in the source. One or more listeners register interest (subscribe) to the stream of events, providing a function to be called when a new event occurs. In this case, the mouse is the event source, and the events are changes in the state of the mouse: its x,y position or the state of its buttons (whether they are pressed or released). Events often include additional information about the transition (such as the x,y position of mouse), which might be bundled into an **event object** or passed as parameters.

When an event occurs, the event source distributes it to all subscribed listeners, by calling their callback functions.

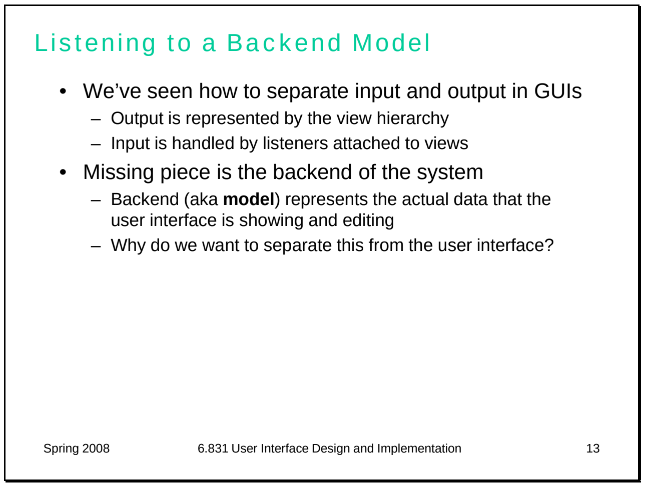


Other Examples of Observer

- Higher-level GUI input events
  - A Button sends an action event when it is pressed (whether by the mouse or by the keyboard)
  - A Textbox sends change events when its contents change
- Internet messaging
  - Email mailing lists
  - IM chatrooms

Low-level mouse and keyboard handling isn’t the only way the Observer pattern is used in GUIs. Many listeners in a view hierarchy may be watching for higher-level events. For example, pressing a GUI button triggers a high-level activation event (sometimes called an action event or a command event). It’s better to listen for this high-level event, rather than a mouse click event, because a button can be triggered by the keyboard as well as by the mouse. Similarly, a textbox sends events when its state changes, regardless of what caused the change.

Observer patterns can be found in higher-level communication systems too – a mailing list or IM chatroom is basically a stream of events with a set of subscribers.

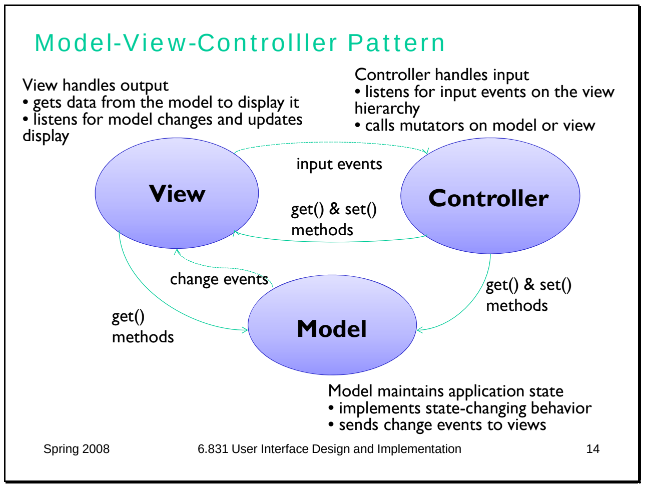


Listening to a Backend Model

- We’ve seen how to separate input and output in GUIs
  - Output is represented by the view hierarchy
  - Input is handled by listeners attached to views
- Missing piece is the backend of the system
  - Backend (aka **model**) represents the actual data that the user interface is showing and editing
  - Why do we want to separate this from the user interface?

We’ve seen how GUI programs are structured around a view hierarchy, and how input events are handled by attaching listeners to views. This is the start of a separation of concerns – output handled by views, and input handled by listeners.

But we’re still missing the application itself – the backend that actually provides the information to be displayed, and computes the input that is handled.



Model-View-Controlller Pattern

View handles output
• gets data from the model to display it
• listens for model changes and updates display

Controller handles input
• listens for input events on the view hierarchy
• calls mutators on model or view

Model maintains application state
• implements state-changing behavior
• sends change events to views

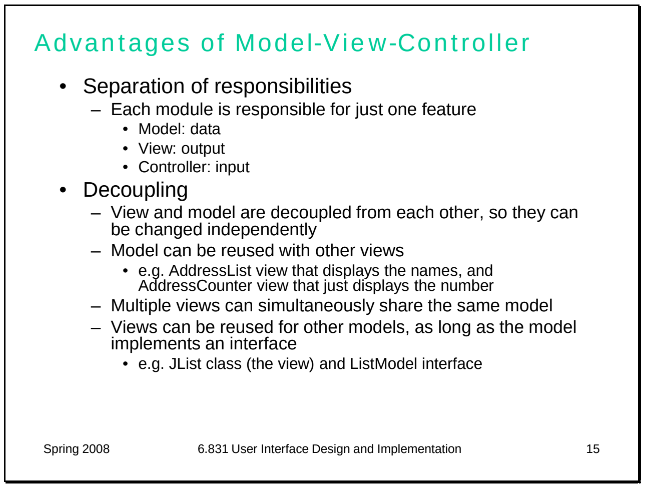
The **model-view-controller** pattern, originally articulated in the Smalltalk-80 user interface, has strongly influenced the design of UI software ever since. In fact, MVC may have single-handedly inspired the software design pattern movement; it figures strongly in the introductory chapter of the seminal "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 change 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.



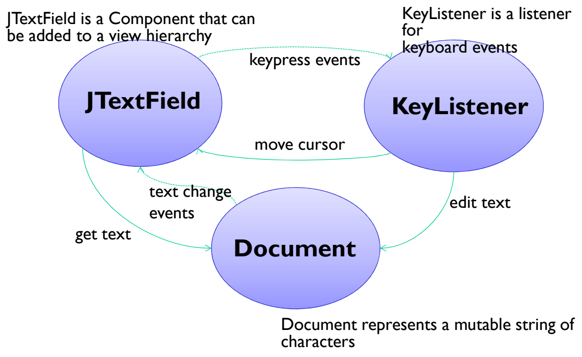
Advantages of Model-View-Controller

- Separation of responsibilities
  - Each module is responsible for just one feature
    - Model: data
    - View: output
    - Controller: input
- Decoupling
  - View and model are decoupled from each other, so they can be changed independently
  - Model can be reused with other views
    - e.g. AddressList view that displays the names, and AddressCounter view that just displays the number
  - Multiple views can simultaneously share the same model
  - Views can be reused for other models, as long as the model implements an interface
    - e.g. JList class (the view) and ListModel interface

Spring 2008 6.831 User Interface Design and Implementation 15

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 models to be reused in other applications. The MVC pattern enables the creation of user interface **toolkits**, which are libraries of reusable interface objects.

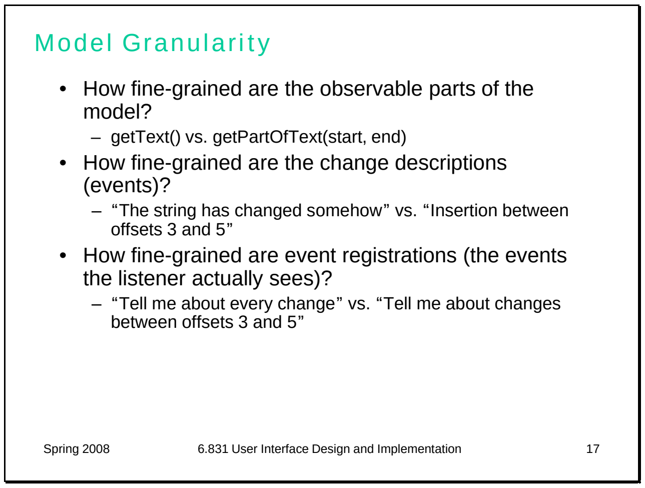
Another MVC Example: Textbox


JTextField is a Component that can be added to a view hierarchy
KeyListener is a listener for keyboard events
keypress events
JTextField
KeyListener
move cursor
text change events
edit text
get text
Document
Document represents a mutable string of characters

Spring 2008 6.831 User Interface Design and Implementation 16

A simple example of the MVC pattern is a text field widget (this is Swing's 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.

Instances of the MVC pattern appear at many scales in GUI software. At a higher level, this text field might be part of a view (like the address book editor), with a different controller listening to it (for text-changed events), for a different model (like the address book). But when you drill down to a lower level, the text field itself is an instance of MVC.



Model Granularity

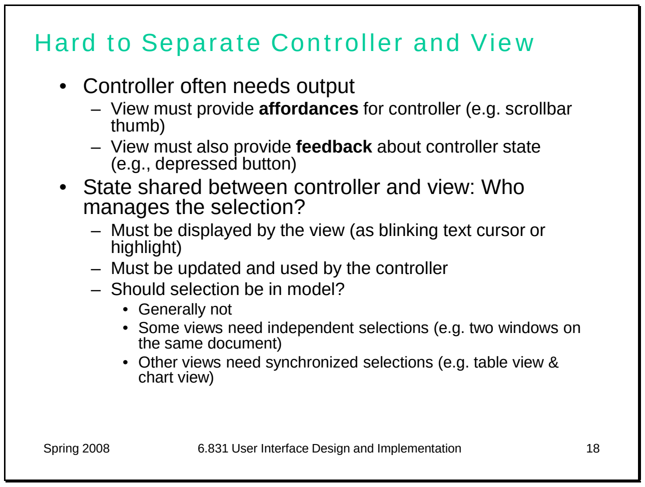
- How fine-grained are the observable parts of the model?
  - getText() vs. getPartOfText(start, end)
- How fine-grained are the change descriptions (events)?
  - “The string has changed somehow” vs. “Insertion between offsets 3 and 5”
- How fine-grained are event registrations (the events the listener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

Spring 2008 6.831 User Interface Design and Implementation 17

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. A mapping application, like Google Maps, is worse – the model contains 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.

Controlling the granularity of notification or registration is crucial to achieving good interactive view performance on large models, like Google Maps.



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)

Spring 2008 6.831 User Interface Design and Implementation 18

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.

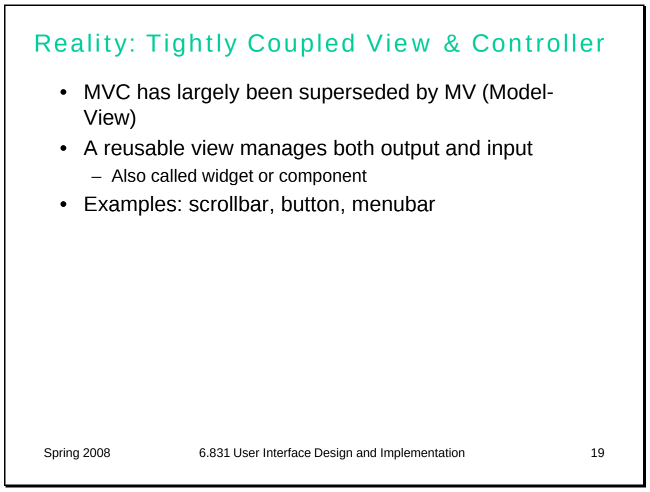
First, a controller often needs to produce its own output. The view must display **affordances** for the controller, such as selection handles or scrollbar thumbs. The controller must be aware of the screen locations of these affordances. When the user starts manipulating, the view must modify its appearance 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.



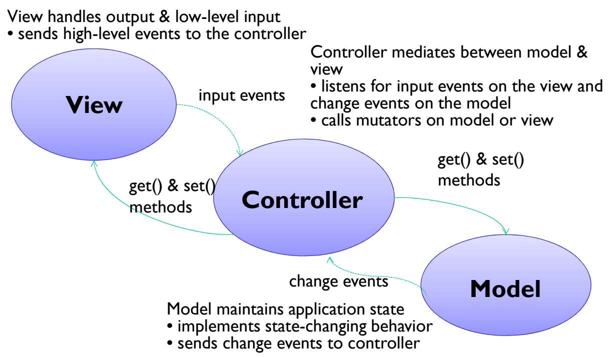
Reality: Tightly Coupled View & Controller

- 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

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.

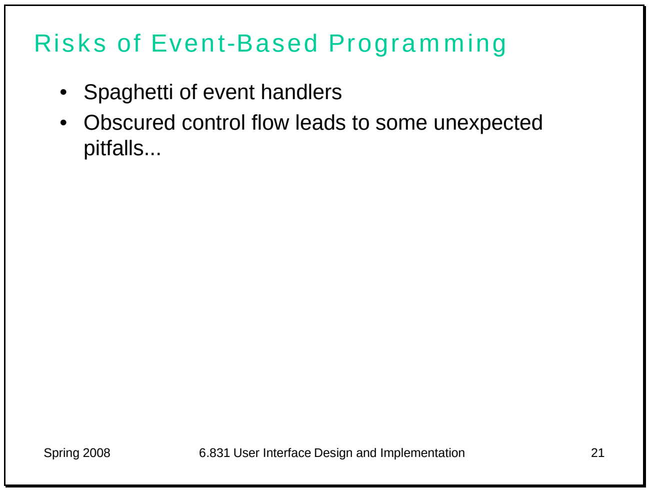
A Different Perspective on MVC


View handles output & low-level input
• sends high-level events to the controller
Controller mediates between model & view
• listens for input events on the view and change events on the model
• calls mutators on model or view
View
input events
Controller
get() & set() methods
get() & set() methods
Model
change events
Model maintains application state
• implements state-changing behavior
• sends change events to controller

Partly in response to this difficulty, and also to provide a better decoupling between the model and the view, some definitions of the MVC pattern treat the controller less as an input handler and more as a **mediator** between the model and the view.

In this perspective, the view is responsible not only for output, but also for low-level input handling, so that it can handle the overlapping responsibilities like affordances and selections.

But listening to the model is no longer the view's responsibility. Instead, the controller listens to both the model and the view, passing changes back and forth. The events receiving high-level input events from the view, like selection-changed, button-activated, or textbox-changed, rather than low-level input device events).



Risks of Event-Based Programming

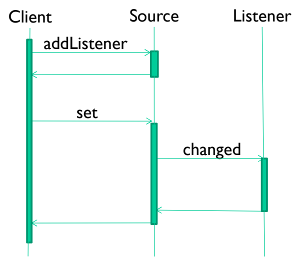
- Spaghetti of event handlers
- Obscured control flow leads to some unexpected pitfalls...

Spring 2008 6.831 User Interface Design and Implementation 21

Whichever pattern you prefer, it's very important to structure your GUI program carefully. Control flow through an event-based program is not simple. You can't follow the control just by studying the source code**,** because control flow depends on listener relationships established at runtime, and input events happening nondeterministically. Careful discipline about who listens to what (like the model-view-controller pattern) is essential for limiting the complexity of control flow and understanding how to debug your program.

The hidden control flow leads to some unexpected pitfalls, which is the last thing we'll look at in today's lecture.

Basic Interaction of Event Passing



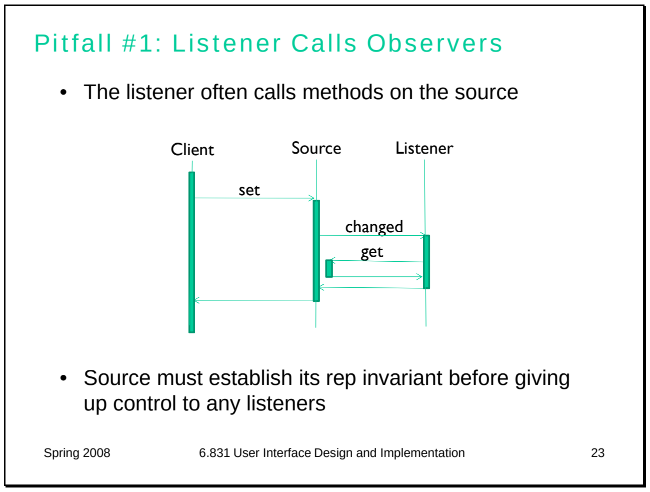
```
interface Source {
    addListener()
    removeListener()
    get()
    set()
}

interface Listener {
    changed()
}
```

Spring 2008 6.831 User Interface Design and Implementation 22

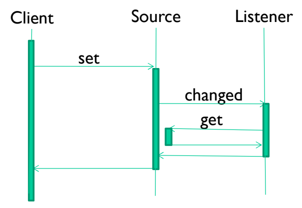
First, a bit of notation. This is a **sequence diagram**, which is useful for depicting control flow. Time flows downward. Vertical time lines represent objects, such as an event source or a listener. Horizontal arrows show method calls and returns passing control between objects. Finally, dark rectangles show when a method is active (i.e., on the stack).

Here's the conventional interaction that occurs in the observer pattern. A client uses **addListener** (or a similar method) registers a listener to receive notifications from the event source. Then, when the source changes state (usually due to some other object calling a mutator method, like a **set** method), it fires an event to all its registered listeners by calling **changed** on them.



Pitfall #1: Listener Calls Observers

- The listener often calls methods on the source

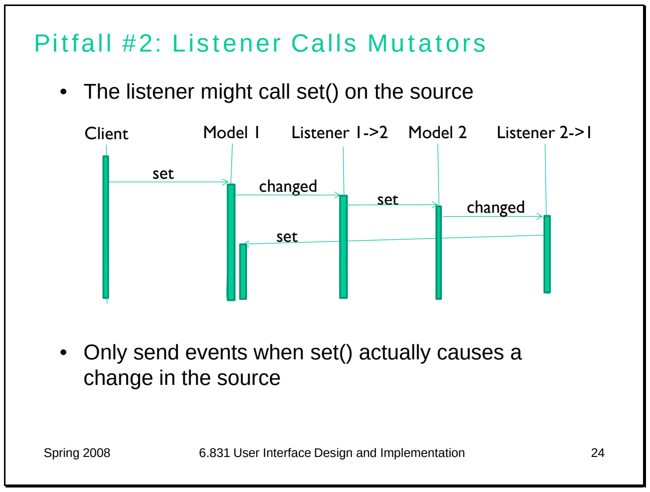


- Source must establish its rep invariant before giving up control to any listeners

This leads to the first pitfall. The listener often reacts to the change in the model by pulling more data from the source using **get**() calls. For example, when a textbox gets a change event from its model, it needs to call getText() to get the new text and display it. So calls to get() may occur while set() is still in progress.

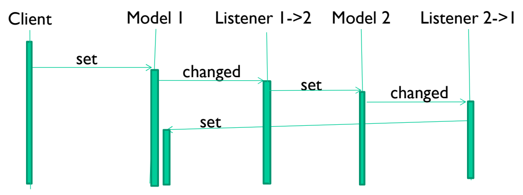
Why is this a potential problem? Because the set() method hasn't returned yet, it's possible that the source data structure is not yet in a consistent state, causing the get() method to return garbage (or worse, throw an exception). When the source calls changed() on its listeners, it is giving up control – in much the same way that a method gives up control when it returns to its caller. So the source 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 listeners.

It's often best to delay firing off events until the end of the method that caused the modification. Don't fire events while you're in the midst of making changes to the model's data structure.



Pitfall #2: Listener Calls Mutators

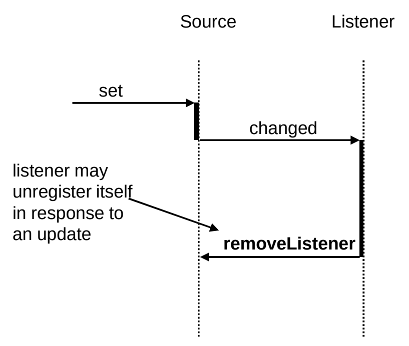
- The listener might call set() on the source



- Only send events when set() actually causes a change in the source

Another pitfall occurs when an observer responds to an update message by calling **set** on the model. Why would it do that? It might, for instance, be trying to keep the model within some legal range. Or two models could be listening to each other in order to keep their state synchronized. So calls to set() may occur while set() is still in progress. Obviously, this could lead to infinite regress if you're not careful. A good practice for models to protect themselves against this regress is to only send updates if a change actually occurs; if a client calls set() but it has no actual effect on the model, then no updates should be sent.

Pitfall #3: Listener Unregisters During Update



Another potential pitfall is a listener that unregisters itself with **removeListener**. For example, suppose we have a model of stock market data, and a listener that's watching for a certain stock to reach a certain price. Once the stock hits the target price, the listener does its thing (e.g., popping up a window to notify the user, or executing a trade); 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 listeners, 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.

Summary

- View hierarchy
  - Primary structuring pattern for GUI programs
  - Used for output, input, and layout
- Observer pattern
  - Used for low-level mouse and keyboard input handling
  - Also high-level input events and model changes
  - Beware the pitfalls
- Model-view-controller pattern
  - Decouples backend from user interface
  - Aims to decouple output from input, but that's hard to do in practice
  - Controller may become a mediator instead