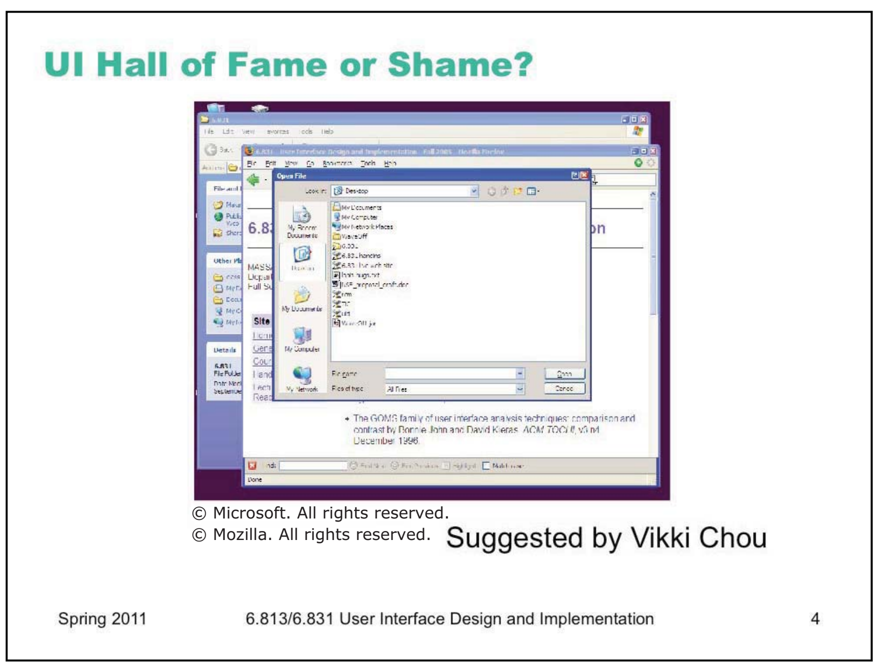
# Lecture 9: UI Software Architecture

RS1 (for 6.831G), PS1 (for 6.813U)
released today, due Sunday

Content in this lecture indicated as "All Rights Reserved" is excluded from our Creative Commons license. For more information, see http://ocw.mit.edu/fairuse.

## UI Hall of Fame or Shame?



© Microsoft. All rights reserved.
© Mozilla. All rights reserved.
Suggested by Vikki Chou

Spring 2011 6.813/6.831 User Interface Design and Implementation 4

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.

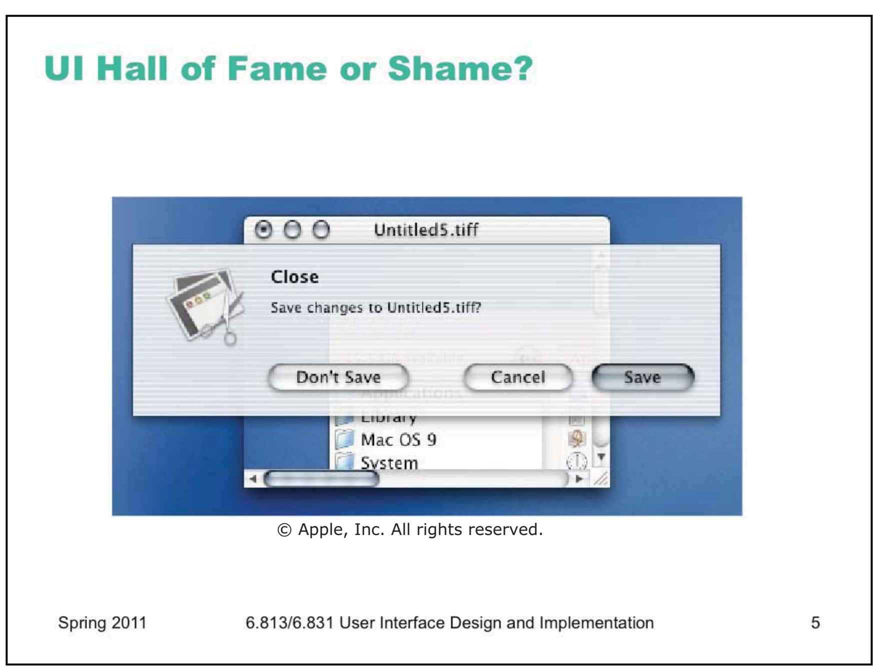
Modal dialogs do have some usability advantages, such as **error prevention** (the modal dialog is always on top, so it can't get lost or be ignored, and the user can't accidentally change the selection in the main window while working on a modal dialog that affects that selection).

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.




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.

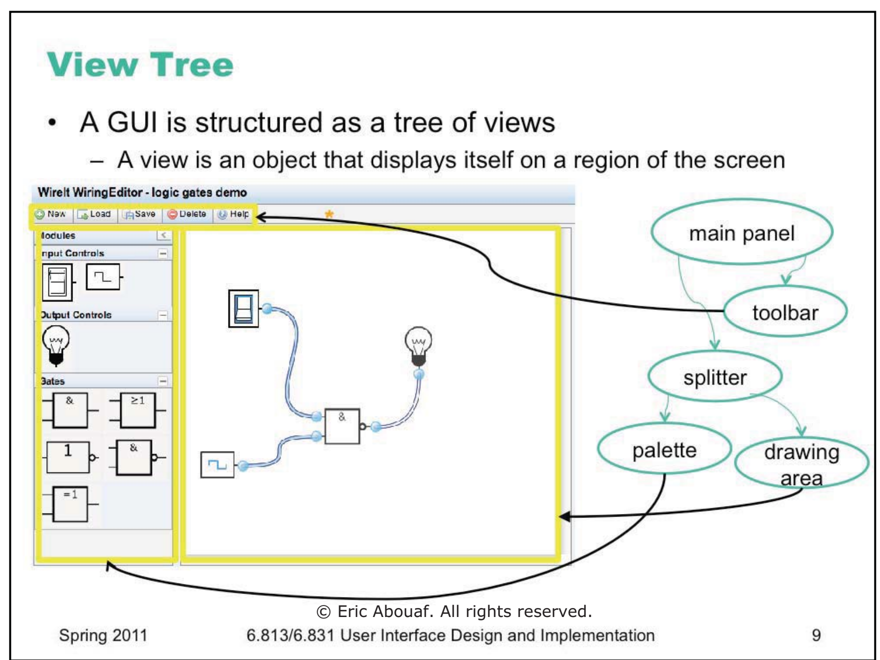
## Today's Topics

- Design patterns for GUIs
  - View tree
  - Listener
  - Widget
  - Model-view-controller
- Approaches to GUI programming
  - Procedural
  - Declarative
  - Direct manipulation
- Web UI at lightning speed
  - HTML
  - Javascript
  - jQuery



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 tree**, which is a central feature in the architecture of every important GUI toolkit; and the **listener** pattern, which is essential to decoupling the model from the view and controller.

We'll also look at the three main approaches to implementing GUIs, and use that context for a quick introduction to HTML, Javascript, and jQuery, which together with CSS (next lecture) constitute the user interface toolkit that we'll be using in lectures and problem sets in this class. Note that the backend development of web applications falls outside the scope of the course material in this class. So we won't be talking about things like SQL, PHP, Ruby on Rails, or even AJAX. For more about that, you may want to check out the 6.470 IAP web programming competition, or the soon-to-be-offered 6.170 web programming software lab.




This leads to the first important pattern we'll talk about today: the **view tree.** 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 or nodes; in other toolkits, they may be called widgets, controls, or interactors.

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

## How the View Tree Is Used

- Output
  - GUIs change their output by **mutating** the view tree
  - A redraw algorithm automatically redraws the affected views
- Input
  - GUIs receive keyboard and mouse input by attaching listeners to views (more on this in a bit)
- Layout
  - Automatic layout algorithm traverses the tree to calculate positions and sizes of views

Spring 2011 6.813/6.831 User Interface Design and Implementation 10

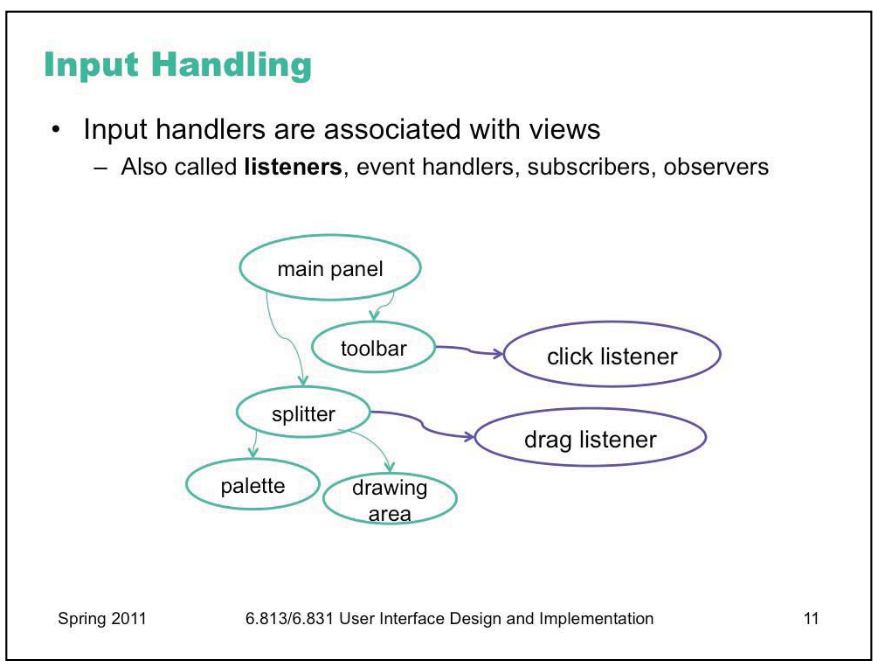
Virtually every GUI system has some kind of view tree. The view tree is a powerful structuring idea, which is loaded with responsibilities in a typical GUI:

**Output.** Views are responsible for displaying themselves, and the view hierarchy directs the display process. GUIs change their output by mutating the view tree. For example, in the wiring diagram editor shown on the previous slide, the wiring diagram is changed by adding or removing objects from the subtree representing the drawing area. A redraw algorithm automatically redraws the affected parts of the subtree.

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

**Layout.** The view tree controls how the views are laid out on the screen, i.e. how their bounding boxes are assigned. An automatic layout algorithm automatically calculates positions and sizes of views.

We'll look at more about each of these areas in the next three lectures.




To handle mouse input, for example, we can attach a handler to the view that is called when the mouse is clicked on it. Handlers are variously called **listeners**, event handlers, subscribers, and observers.

## Listener Pattern

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

Spring 2011 6.813/6.831 User Interface Design and Implementation 12

GUI input event handling is an instance of the Listener pattern (also known as Observer and Publish-Subscribe). In the Listener 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.

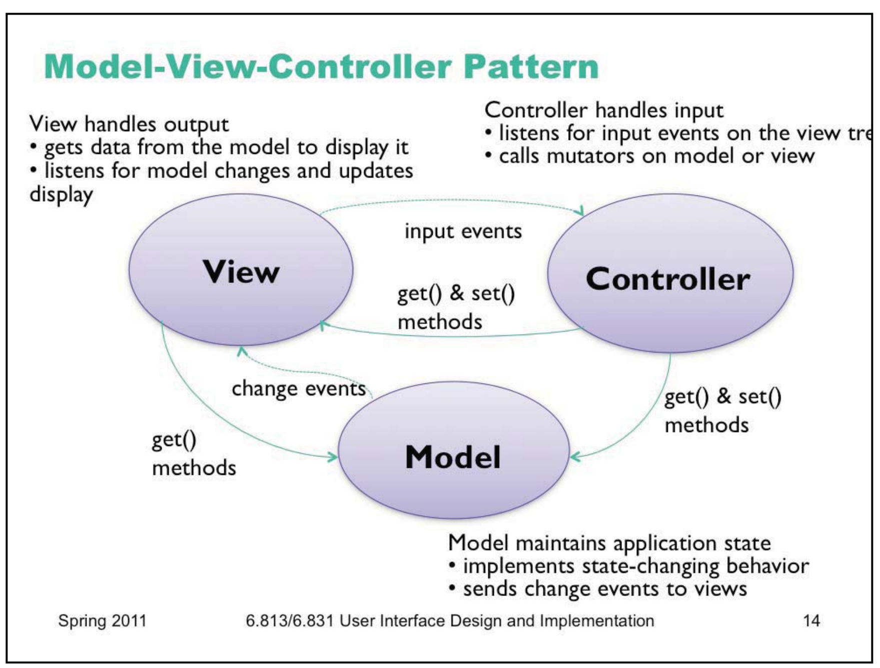
## Separating Frontend from Backend

- We've seen how to separate input and output in GUIs
  - Output is represented by the view tree
  - 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 tree, 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.




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 **listener 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.

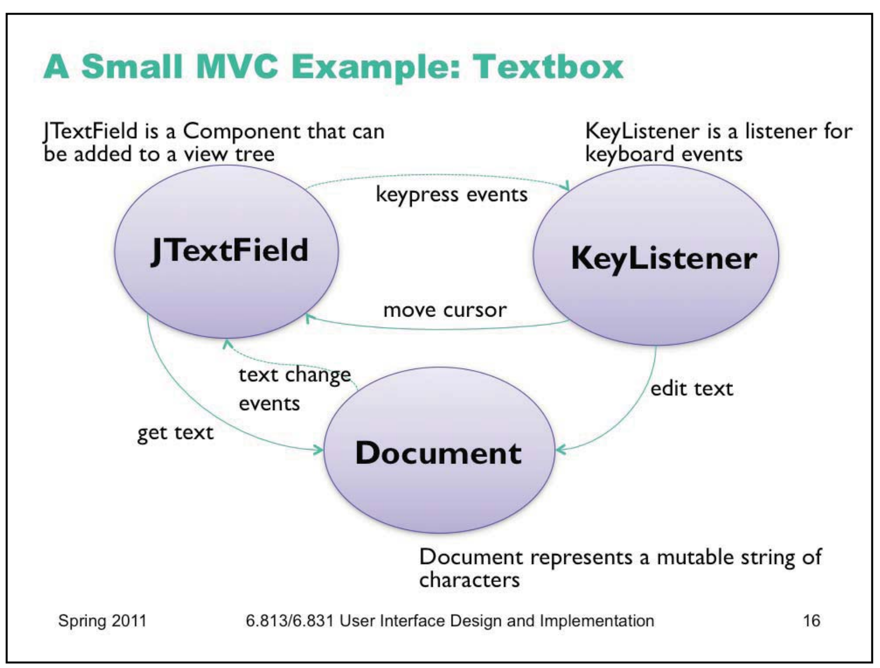
Finally, the controller handles the input. It receives keyboard and mouse events, and instructs the model to change accordingly.

## 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
  - Multiple views can simultaneously share the same model
  - Views can be reused for other models, as long as the model implements an interface

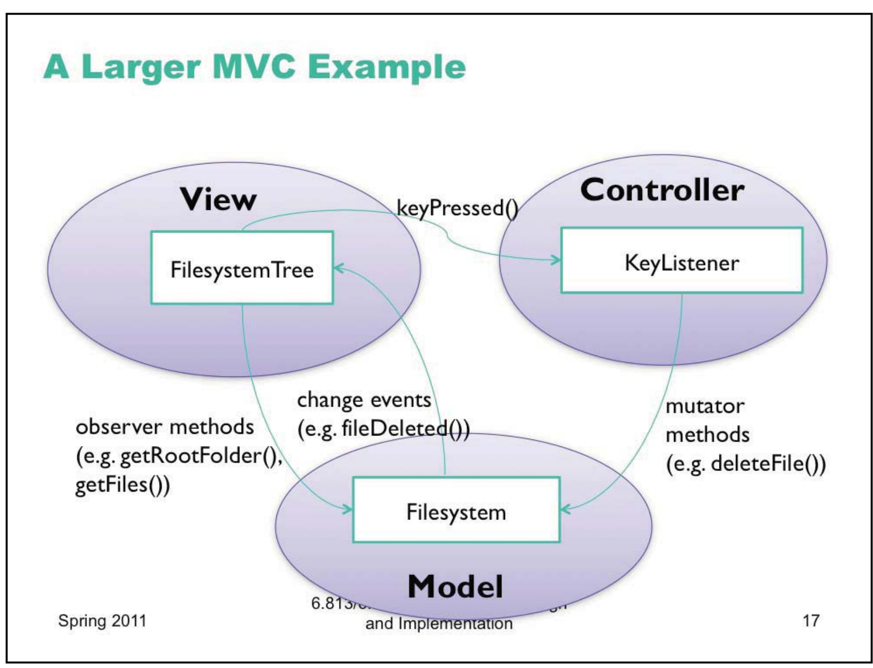


In principle, this separation has several benefits. First, it allows the interface to have multiple views showing the same application data. For example, a database field might be shown in a table and in an editable form at the same time. Second, it allows views and models to be reused in other applications.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




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




Here's a larger example, in which the view is a filesystem browser (like the Mac Finder or Windows Explorer), the model is the disk filesystem, and the controller is an input handler that translates the user's keystrokes and mouse clicks into operations on the model and view.

## 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 2011 6.813/6.831 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.

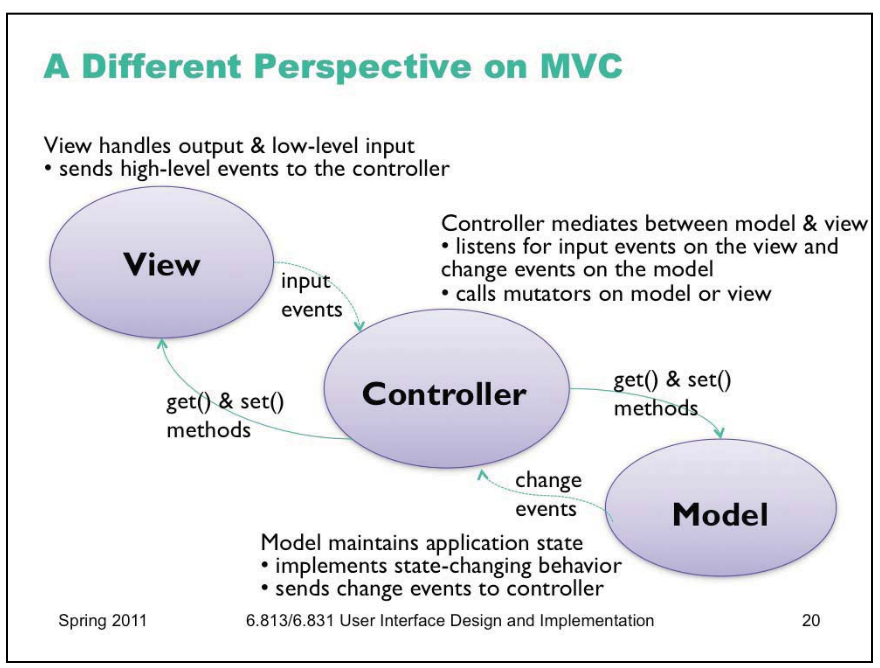
## Widget: Tightly Coupled View & Controller

- The MVC idea has largely been superseded by an MV (Model-View) idea
- A widget is a reusable view object that manages both its output and its input
  - Widgets are sometimes called components (Java, Flex) or controls (Windows)
- 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 a GUI toolkit are fused view/controllers like this; you can't, for example, pull out the scrollbar's controller and reuse it in your own custom scrollbar. Internally, the scrollbar probably follows a model-view-controller architecture, but the view and controller aren't independently reusable.

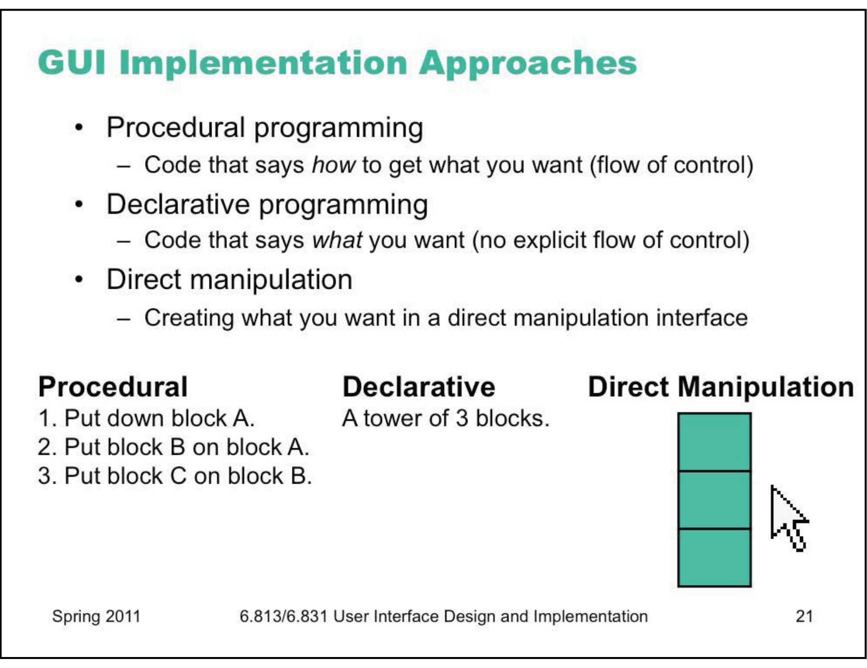



Partly in response to this difficulty, and also to provide a better decoupling between the model and the view, some definitions of the MVC pattern treat the controller less as an input handler and more as a **mediator** between the model and the view.

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).

The Mac Cocoa framework uses this approach to MVC.



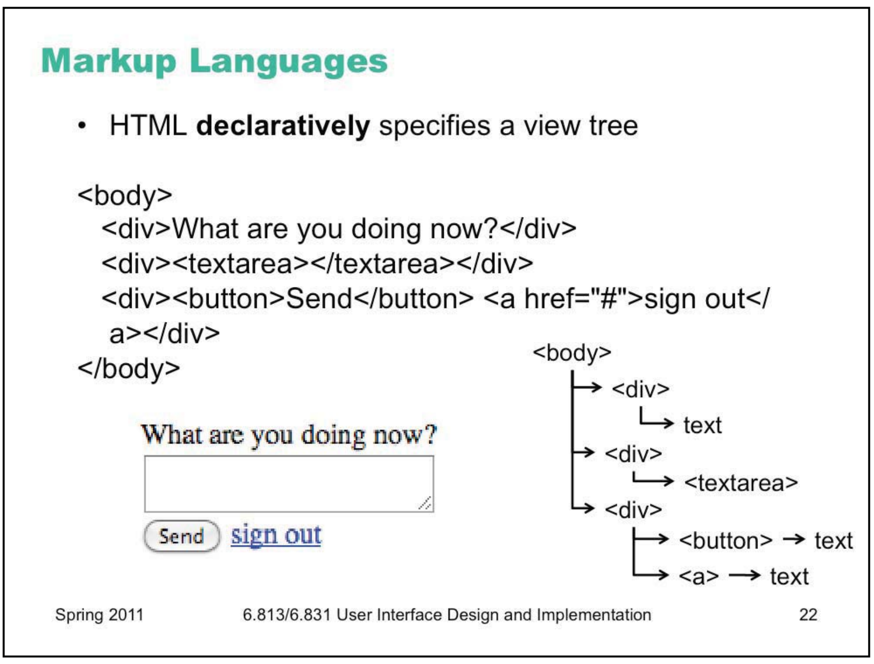

Now let's talk about how to construct the view tree, which will be a tale of three paradigms.

In **procedural** style, the programmer has to say, step-by-step, how to reach the desired state. There's an explicit thread of control. This means you're writing code (in, say, Javascript or Java) that calls constructors to create view objects, sets properties of those objects, and then connects them together into a tree structure (by calling, say, appendChild() methods). Java Swing programming was largely procedural. Virtually every GUI toolkit offers an API like this for constructing and mutating the view tree.

In **declarative** style, the programmer writes code that directly represents the desired view tree. There are many ways to describe tree structure in textual syntax, but the general convention today is to use an HTML/XML-style markup language. There's no explicit flow of control in a declarative specification of a tree; it doesn't *do*, it just *is*. An automatic algorithm translates the declarative specification into runtime structure or behavior.

Finally, in **direct manipulation** style, the programmer uses a direct-manipulation graphical user interface to create the view tree. These interfaces are usually called GUI builders, and they offer a palette of view object classes, a drawing area to arrange them on, and a property editor for changing their properties.

All three paradigms have their uses, but the sweet spot for GUI programming basically lies in an appropriate mix of declarative and procedural – which is what HTML/Javascript provides.




Our first example of declarative UI programming is a **markup language**, such as HTML. A markup language provides a declarative specification of a view hierarchy. An HTML **element** is a component in the view hierarchy. The type of an element is its **tag**, such as div, button, and img. The properties of an element are its **attributes**. In the example here, you can see the id attribute (which gives a unique name to an element) and the src attribute (which gives the URL of an image to load in an img element); there are of course many others.

There's an automatic algorithm, built into every web browser, that constructs the view hierarchy from an HTML specification – it's simply an HTML parser, which matches up start tags with end tags, determines which elements are children of other elements, and constructs a tree of element objects as a result. So, in this case, the automatic algorithm for this declarative specification is pretty simple.

## HTML Syntax

```
<!DOCTYPE html>
<html>
<head></head>
<body>
     <button disabled="true">
          <img src='dont.png' />
          Don't Press Me
     </button>
               <!--
                         <button>
                            Press Me Instead
                         </button>
               -->
</body>
</html>
```



Boilerplate: DOCTYPE, html, head, and body elements should be part of every HTML file.

An element consists of a start tag, attributes, content, and end tag.

Case doesn’t matter for tag names and attribute names

Attribute values can be ‘quoted’ or “quoted”

    or not quoted at all, but it’s better to quote

Text outside of a tag is grouped together into a “text node”

Whitespace is (mostly) ignored

Some kinds of elements are void (never have an end tag)

    e.g. img, br

    this is often reinforced with an extra slash: <img src=“…” />, both to help the reader, and because XML parsers demand it before they’ll consider your HTML file (HTML is related to XML, and occasionally they try to play nicely together)

Comments look like <!--     -->

<, >, and & need to be escaped: < > & respectively

## Important HTML Elements for UI Design

- **Layout**
  - Box <div>
  - Grid <table>, <tr>, <td>
- **Text**
  - Font & color <span>
- **Widgets**
  - Hyperlink <a>
  - Button <button>
  - Textbox <input type="text">
  - Multiline text <textarea>
  - Rich text <div contenteditable="true">
  - Drop-down <select> <option>
  - Listbox <select multiple="true">
  - Checkbox <input type="checkbox">
  - Radiobutton <input type="radio">
- **Pixel output**
  - <img>
- **Stroke output**
  - <canvas> (Firefox, Safari)
- **Javascript code**
  - <script>
- **CSS style sheets**
  - <style>



Here is a cheat sheet of the most important elements that you might use in an HTML-based user interface.

The **<div>** and **<span>** elements are particularly important, and may be less familiar to people who have only used HTML for writing textual web pages. By default, these elements have no presentation associated with them; you have to add it using style rules (which we'll explain next lecture). The <div> element creates a box, and the <span> element changes textual properties like font and color while allowing its contents to flow and word-wrap.

HTML has a rather limited set of widgets. There are other declarative UI languages similar to HTML that have much richer sets of built-in components, such as MXML (used in Adobe Flex) and XUL (used in Mozilla Firefox) and XAML (used in Microsoft WPF and Silverlight).

We'll talk more about the output elements, img and canvas, in the output lecture.

The <script> element to embed procedural code (usually Javascript) into an HTML specification. This actually breaks the model of declarative programming, because it introduces an explicit flow of control! The <script> elements are executed in the order that they are encountered in parsing the HTML, which means that they might see only a partially-constructed tree.

Finally, the <style> element is used for embedding another declarative specification, CSS style sheets, which we'll look at next lecture.

## View Tree Manipulation

- Javascript can **procedurally** mutate a view tree

```
<script>
var doc = document
var div1 = doc.createElement("div")
  div1.appendChild(doc.createTextNode("What are you doing now?”)
…
var div3 = doc.createElement("div")
  var button = doc.createElement("button")
    button.appendChild(doc.createTextNode("Send"))
    div3.appendChild(button)
  var a = doc.createElement("a")
    a.setAttribute("href", "#")
    a.appendChild(doc.createTextNode("sign out"))
    div3.appendChild(a)
</script>
```

What are you doing now?

Send sign out

Spring 2011 6.813/6.831 User Interface Design and Implementation 25

Here’s procedural code that generates the same HTML view tree, using Javascript and the Document Object Model (DOM). DOM is a standard set of classes and methods for interacting with a tree of HTML or XML objects procedurally. DOM interfaces exist not just in Javascript, which is the most common place to see it, but also in Java and other languages.

Note that the name DOM is rather unfortunate from our point of view. It has nothing to do with “models” in the sense of model-view-controller – in fact, the DOM is a tree of *views*. It’s a model in the most generic sense we discussed in the Learnability lecture, a set of parts and interactions between them, that allows an HTML document to be treated as objects in an object-oriented programming language.

Most people ignore what DOM means, and just use the word (pronouncing it “Dom” as in “Dom DeLouise”). In fact DOM is often used to refer to the view tree.

Compare the procedural code here with the declarative code earlier.

Incidentally, you don’t always have to use the setAttribute method to change attributes on HTML elements. In Javascript, many attributes are reflected as properties of the element (analogous to fields in Java). For example, **obj.setAttribute(“id”, value)** could also be written as **obj.id = value**. Be warned, however, that only standard HTML attributes are reflected as object properties (if you call setAttribute with your own wacky attribute name, it won’t appear as a Javascript property), and sometimes the name of the attribute is different from the name of the property. For example, the “class” attribute must be written as obj.className when used as a property.

Raw DOM programming is painful, and worth avoiding. Instead, there are toolkits that substantially simplify procedural programming in HTML/Javascript -- jQuery is a good example, and the one we’ll be using.

## Javascript in One Slide

### Like Java…

**expressions**

```
hyp = Math.sqrt(a*a + b*b)
console.log("Hello"
            + ", world");
```

**statements**

```
if (a < b) { return a }
   else { return b }
```

**comments**

```
/* this
    is a comment */
// and so is this
```

### Like Python…

**no declared types**

```
var x = 5;
for (var i = 0; i < 10; ++i) {…}
```

**objects and arrays are dynamic**

```
var obj = { x: 5, y: -1 };
obj.z = 8;
var list = ["a", "b"];
list[2] = "c";
```

**functions are first-class**

```
function square(x) { return x*x; }
var double = function(a) {
                 return 2*a; }
```

Spring 2011 6.813/6.831 User Interface Design and Implementation 26

Here's everything you need to know about Javascript. Ha! Not exactly. But Javascript is not a hard language to pick up – it's a lot like Java and Python in many ways, and you probably already know Java and Python. Most of the differences are syntactic, which is visible and easy to learn by example. The trickiest pitfalls in Javascript (or in learning any language) are its semantics. Javascript's particular semantic pitfalls are **variable scoping** (which unlike Java is *function* scoped, not block scoped,and unlike Python it defaults to putting new variables in the *global* scope rather than the local scope) and the semantics of **this** (which doesn't behave quite like Java's this or Python's self). The variable scoping pitfalls are responsible for both warnings on this slide – (a) never omit the var keyword when you introduce a new variable, and (b) even though you should use var in your for loops, don't expect it to behave as in Java – there's only one variable i for the entire function, it isn't just scoped to the body of the for loop. A corollary of that is that functions you create within the body of the for loop all share the same variable i. (See "The Infamous Loop Problem" in http://robertnyman.com/2008/10/09/explaining-javascript-scope-and-closures/)

A good online tutorial for Javascript is "A re-introduction to JavaScript" (https://developer.mozilla.org/en/JavaScript/A_re-introduction_to_JavaScript).

Some good online articles describing the pitfalls of scoping and this:

- http://jszen.blogspot.com/2005/04/variable-scoping-gotchas.html
- http://stackoverflow.com/questions/500431/javascript-variable-scope
- http://robertnyman.com/2008/10/09/explaining-javascript-scope-and-closures/
- http://www.digital-web.com/articles/scope_in_javascript/

## jQuery in One Slide

- Select nodes

```
$("#send")
$(".toolbar")
$("button")
```

```
<button id="send" class="toolbar">
  Send
</button>
```

- Create nodes

```
$('<button class="toolbar"></button>')
```

- Act on nodes

```
$("#send").text()            // returns "Send"
$("#send").text("Tweet")     // changes button label
$(".toolbar").attr("disabled", "true")
$("#send").click(function() { … })
$("#textarea").val()
$("#mainPanel").html("<button>Press Me</button>")
```

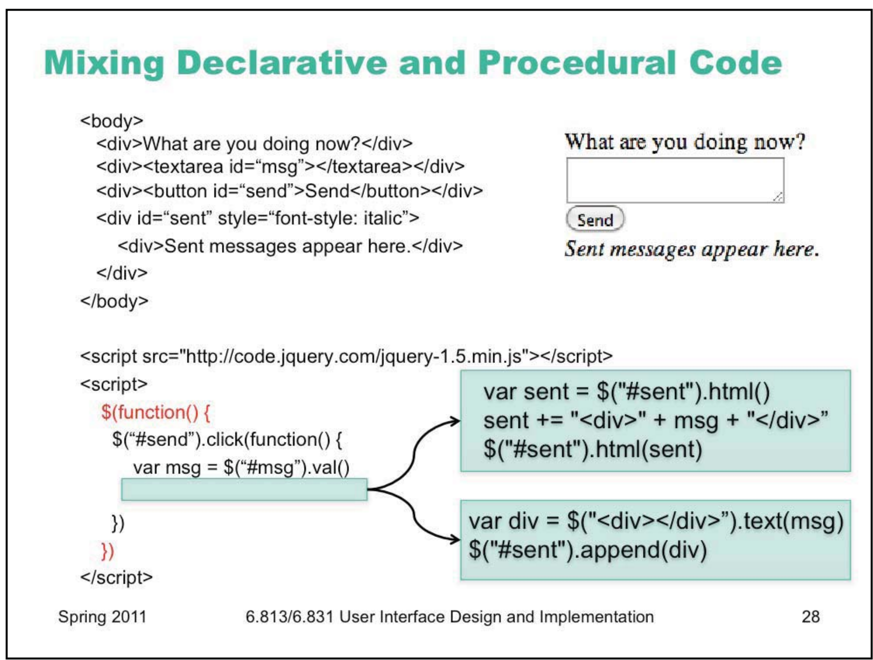


jQuery offers a much better way to interact with the DOM than the actual DOM interface. jQuery is a Javascript library that you include in your HTML page. See jquery.com for more details, documentation, and tutorials.

The essence of jQuery is **selecting** a node (or set of nodes) in the DOM and **acting** on it (getting properties, setting properties, or changing tree structure).

Selection is done by a pattern language (which is a good pattern language to know because it's used in CSS as well, which we'll be learning about in the next lecture). For example, the pattern **#send** finds a node with the id attribute "send", **.toolbar** finds nodes with the class attribute "toolbar", and **button** just finds all <button> nodes.

jQuery provides a variety of methods for acting on the nodes you find. In general, jQuery methods come in pairs with the same name: the method with no arguments gets a value, and the method with arguments sets a value. So .**text()** returns the text contained in the node's descendents, while **.text ("Tweet")** replaces all those descendents with the text node "Tweet". Similarily, **.attr()** gets and sets attribute values, **.click()** sets a mouse event handler (or simulates a click), .**val()** gets or sets the value of a text widget, and .**html()** gets or sets the descendents of a node as HTML.




To actually create a working interface, you frequently need to use a mix of declarative and procedural code. The declarative code is generally used to create the static parts of the interface, while the procedural code changes it dynamically in response to user input or model changes. Even inside the procedural code, we can use declarative code – a template of HTML that is filled with dynamically-computed parts.

One issue to think about is whether this template is constructed as a string of characters (as in the top green box), or as a data structure of objects (as in the bottom green box). Which do you think is better?

Note also that the code in the <script> tag is wrapped in a mysterious $(function() {…}), which is highlighted in red. This is jQuery shorthand for $(document).ready(function() {…})), which is in fact an event handler attached to the root of the view tree (the *document*). This event handler is called just once, after the entire HTML file has been parsed and the tree has been constructed. This is important to do! Why? Where could we put the <script> element so that the Send button doesn't even exist when the <script> element is executed? This is one of the ways that it's tricky to combine procedural and declarative programming.

## Advantages & Disadvantages of Declarative UI

- Usually more compact
- Programmer only has to know how to say *what*, not *how*
  - Automatic algorithms are responsible for figuring out how
- May be harder to debug
  - Can't set breakpoints, single-step, print in a declarative specification
  - Debugging may be more trial-and-error
- Authoring tools are possible
  - Declarative spec can be loaded and saved by a tool; procedural specs generally can't

Spring 2011 6.813/6.831 User Interface Design and Implementation 29

Now that we've worked through our first simple example of declarative UI – HTML – let's consider some of the advantages and disadvantages.

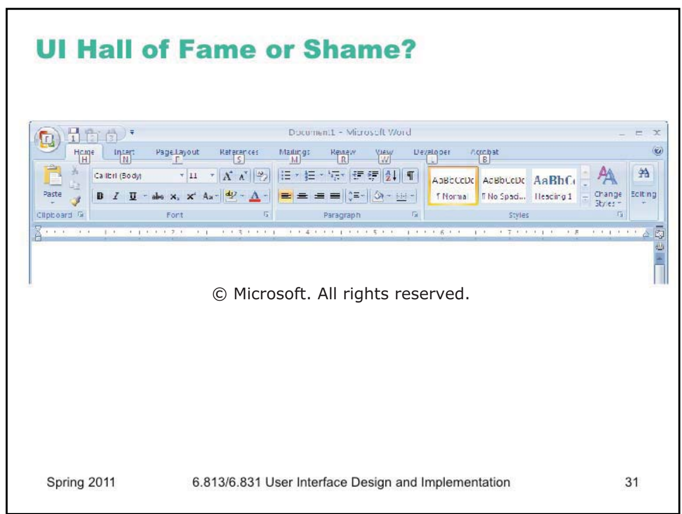
First, the declarative code is usually more compact than procedural code that does the same thing. That's mainly because it's written at a higher level of abstraction: it says *what* should happen, rather than *how*.

But the higher level of abstraction can also make declarative code harder to debug. There's generally no notion of time, so you can't use techniques like breakpoints and print statements to understand what's going wrong. The automatic algorithm that translates the declarative code into working user interface may be complex and hard to control – i.e., small changes in the declarative specification may cause large changes in the output. Declarative specs need debugging tools that are customized for the specification, and that give insight into how the spec is being translated; without those tools, debugging becomes trial and error.

On the other hand, an advantage of declarative code is that it's much easier to build authoring tools for the code, like HTML editors or GUI builders, that allow the user interface to be constructed by direct manipulation rather than coding. It's much easier to load and save a declarative specification than a procedural specification. Some GUI builders ***do*** use procedural code as their file format – e.g., generating Java code and automatically inserting it into a class. Either the code generation is purely one-way (i.e., the GUI builder spits it out but can't read it back in again), or the procedural code is so highly stylized that it amounts to a declarative specification that just happens to use Java syntax. If the programmer edits the code, however, they may deviate from the stylization and break the GUI builder's ability to read it back in.

## Summary

- Design patterns
  - View tree is the primary structuring pattern for GUIs, used for output, input, and layout
  - Listener is used for input and model-view communication
  - Model-view-controller decouples backend from GUI
- Approaches to GUI programming
  - Procedural, declarative, direct manipulation
  - HTML, Javascript, jQuery




Our Hall of Fame or Shame candidate for next time is the command ribbon, which was introduced in Microsoft Office 2007. The ribbon is a radically different user interface for Office, merging the menubar and toolbars together into a single common widget. Clicking on one of the tabs ("Home", "Insert", "Page Layout", etc) switches to a different ribbon of widgets underneath. The metaphor is a mix of menubar, toolbar, and tabbed pane. Notice how UIs have evolved to the point where new metaphorical designs are riffing on existing *GUI* objects, rather than *physical* objects. Expect to see more of that in the future.

Needless to say, strict **external consistency** has been thrown out the window – Office no longer has a menubar or toolbar. But if we were slavishly consistent, we'd never make any progress in user interface design. Despite the radical change, the ribbon *is* still externally consistent in some interesting ways, with other Windows programs and with previous versions of Office. Can you find some of those ways?

The ribbon is also notable for being designed from careful task analysis. How can you tell?

MIT OpenCourseWare
http://ocw.mit.edu

6.831 / 6.813 User Interface Design and Implementation
Spring 2011

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.