

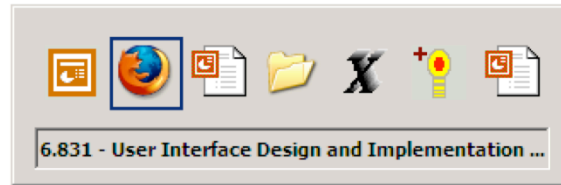
L6: UI Software Architecture

Spring 2013

6.813/6.831 User Interface Design and Implementation

1

UI Hall of Fame or Shame?



Spring 2013

6.813/6.831 User Interface Design and Implementation

2

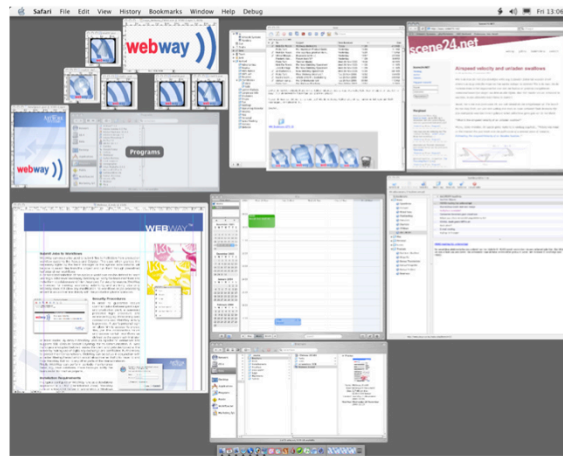
This lecture's candidate for the Hall of Fame & Shame is the **Alt-Tab** window switching interface in Microsoft Windows. This interface has been copied by a number of desktop systems, including KDE, Gnome, and even Mac OS X.

For those who haven't used it, here's how it works. Pressing Alt-Tab makes this window appear. As long as you hold down Alt, each press of Tab cycles to the next window in the sequence. Releasing the Alt key switches to the window that you selected.

We will talk about Alt-Tab from the usability perspective. Think about:

- Is it learnable?
- Is it efficient?
- What about errors and safety?

UI Hall of Fame or Shame?



Spring 2013

6.813/6.831 User Interface Design and Implementation

3

For comparison, we'll also look at the Exposé (now Mission Control) feature in Mac OS X. When you push F3 on a Mac, it displays all the open windows – even hidden windows, or windows covered by other windows – shrinking them as necessary so that they don't overlap. Mousing over a window displays its title, and clicking on a window brings that window to the front and ends the Exposé mode, sending all the other windows back to their old sizes and locations.

Think about the usability of Exposé:

- Is it learnable?
- Is it efficient?
- What about errors and safety?
- How does it compare and contrast with Alt-Tab? Which is more efficient for what tasks, and why?

Today's Topics

- Design patterns for GUIs
 - View tree
 - Listener
 - Model-view-controller
- Approaches to GUI programming
 - Procedural
 - Declarative
 - Direct manipulation

Spring 2013

6.813/6.831 User Interface Design and Implementation

4

Today's lecture is the first in a series 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. We won't get into the details of HTML, CSS, Javascript, and JQuery here: they are well-covered in the labs we hosted earlier this semester (see course web site for materials). Also 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.

VIEW TREE AND THE LISTENER PATTERN

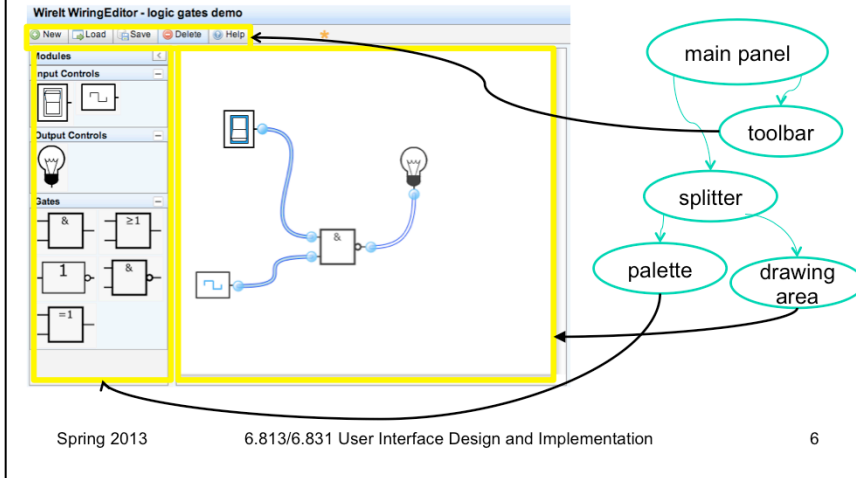
Spring 2013

6.813/6.831 User Interface Design and Implementation

5

View Tree

- A GUI is structured as a tree of views
 - A view is an object that displays itself on a region of the screen



The first important pattern we'll talk about today is 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 aligned with the conceptual structure of the data, so that if a parent is not displayed, its children usually won't be displayed either. In old frameworks (such as Swing), a child was always visually contained (that is, spatially on the 2d display) in its parent's bounding box, but this is not true of the web browser view tree (called the DOM), which allows a designer to position a child independently of the parent.

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 2013

6.813/6.831 User Interface Design and Implementation

7

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.

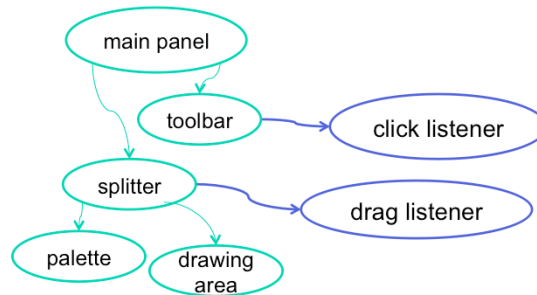
Input. Views can have input handlers, and the view tree controls how mouse and keyboard input is processed.

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 later lectures.

Input Handling

- Input handlers are associated with views
 - Also called **listeners**, event handlers, subscribers, observers



Spring 2013

6.813/6.831 User Interface Design and Implementation

8

- 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 2013

6.813/6.831 User Interface Design and Implementation

9

- 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.

Picoquiz

Select **all** true statements below:

- A. The view tree is only created once and cannot be manipulated while an interface is used.
- B. Each child in a view tree must be spatially contained in the bounding box of its parent in the view tree.
- C. Typically, multiple listeners can be attached to an event.
- D. Listeners allow the GUI to handle input from the user.

Spring 2013

6.813/6.831 User Interface Design and Implementation

10

To answer the picoquiz questions in this lecture, go to:
<http://courses.csail.mit.edu/6.831/2013/picoquiz?lectureId=6>

MODEL-VIEW- CONTROLLER

Spring 2013

6.813/6.831 User Interface Design and Implementation

11

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?

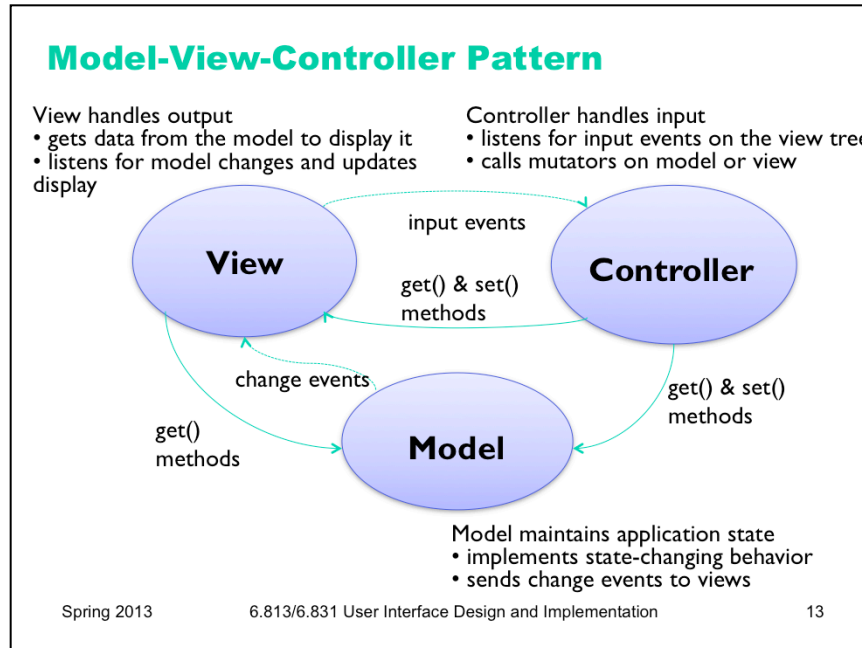
Spring 2013

6.813/6.831 User Interface Design and Implementation

12

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.

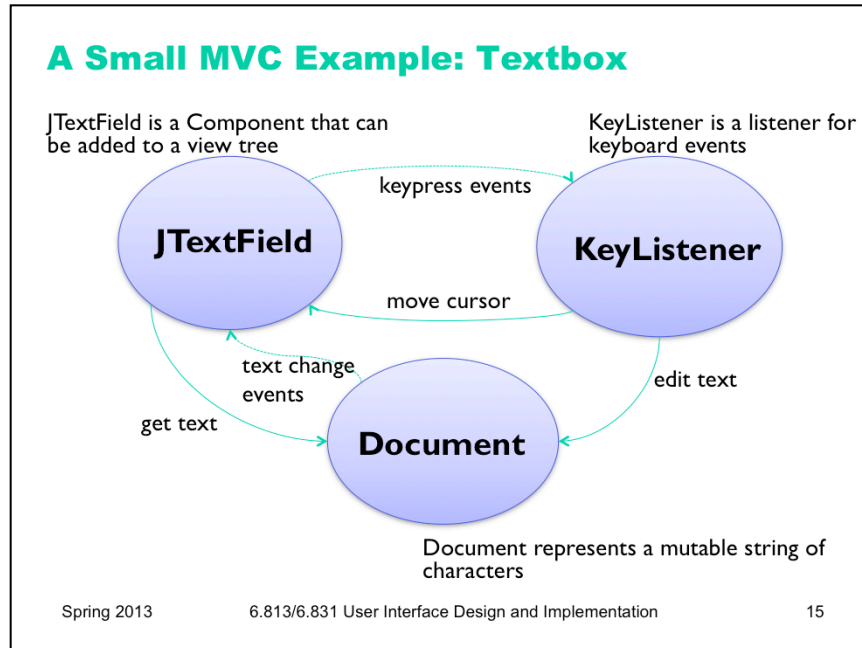
View objects are responsible for output. A view usually occupies some chunk of the screen, usually a rectangular area. Basically, the view queries the model for data and draws the data on the screen. It listens for changes from the model so that it can update the screen to reflect those changes.

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

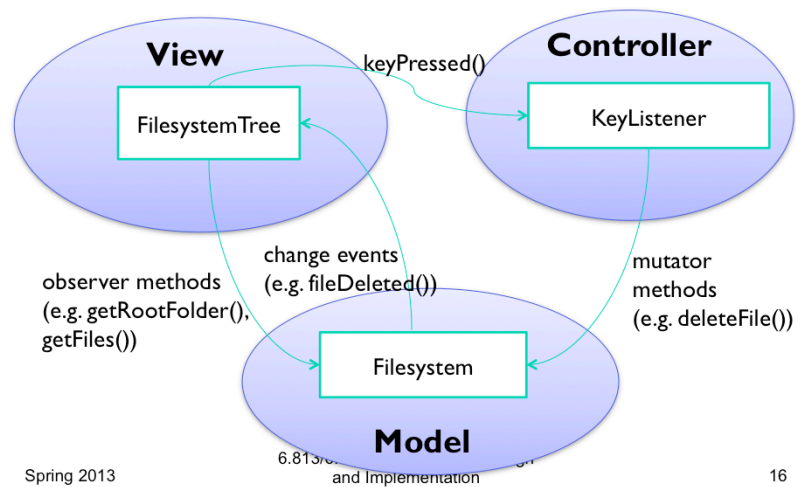
In principle, this separation has several benefits. First, it allows the interface to have multiple views showing the same application data. For example, a database field might be shown in a table and in an editable form at the same time. Second, it allows views and models to be reused in other applications. 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. Note that the controller may signal a change in the view (here, moving the cursor) even when there is no change in the underlying model.

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.

A Larger MVC Example



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 2013

6.813/6.831 User Interface Design and Implementation

17

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

Spring 2013

6.813/6.831 User Interface Design and Implementation

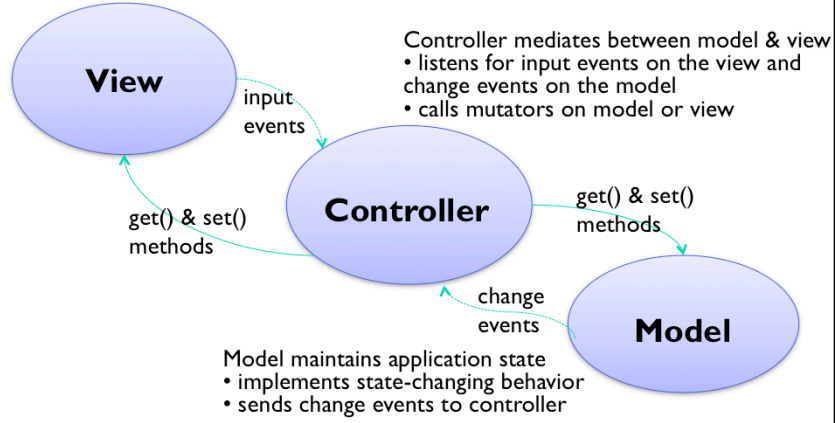
18

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.

A Different Perspective on MVC

View handles output & low-level input
• sends high-level events to the controller



Spring 2013

6.813/6.831 User Interface Design and Implementation

19

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 controller receives 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.

Picoquiz

Model view controller (choose all good answers):

- A. allows the view and the model to change independently.
- B. allows implementing multiple views for a single model.
- C. stores transient UI state (such as selection) in the model.
- D. has been largely superseded by other frameworks that nevertheless maintain a separation between the view and the model.

Spring 2013

6.813/6.831 User Interface Design and Implementation

20

To answer the picoquiz questions in this lecture, go to:
<http://courses.csail.mit.edu/6.831/2013/picoquiz?lectureId=6>

GUI IMPLEMENTATION APPROACHES

Spring 2013

6.813/6.831 User Interface Design and Implementation

21

GUI Implementation Approaches

- Procedural programming
 - Code that says *how* to get what you want (flow of control)
- Declarative programming
 - Code that says *what* you want (no explicit flow of control)
- Direct manipulation
 - Creating what you want in a direct manipulation interface

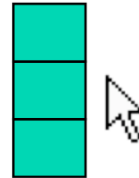
Procedural

1. Put down block A.
2. Put block B on block A.
3. Put block C on block B.

Declarative

A tower of 3 blocks.

Direct Manipulation



Spring 2013

6.813/6.831 User Interface Design and Implementation

22

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) 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). 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.

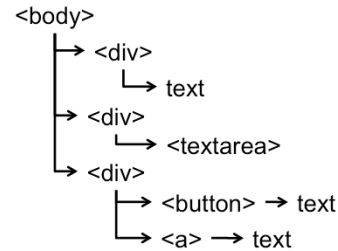
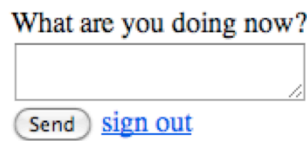
Finally, in **direct manipulation** style, the programmer uses a direct-manipulation graphical user interface to create the view tree. These interfaces are usually called GUI builders, and they offer a palette of view object classes, a drawing area to arrange them on, and a property editor for changing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

Markup Languages

- HTML **declaratively** specifies a view tree

```
<body>
  <div>What are you doing now?</div>
  <div><textarea></textarea></div>
  <div><button>Send</button> <a href="#">sign out</a></div>
</body>
```



Spring 2013

6.813/6.831 User Interface Design and Implementation

23

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.

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?

 [sign out](#)

Spring 2013

6.813/6.831 User Interface Design and Implementation

24

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.

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.

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 2013

6.813/6.831 User Interface Design and Implementation

25

Now that we've seen 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.

Picoquiz

Procedural UI programming (select all good answers):

- A. is typically more compact than declarative programming.
- B. is typically easier to debug than declarative programming.
- C. is typically written in HTML/XML in web applications.
- D. can be used for mutating the view tree in response to user actions.

Spring 2013

6.813/6.831 User Interface Design and Implementation

26

To answer the picoquiz questions in this lecture, go to:
<http://courses.csail.mit.edu/6.831/2013/picoquiz?lectureId=6>

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