UI Software Architecture

Today's Topics

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

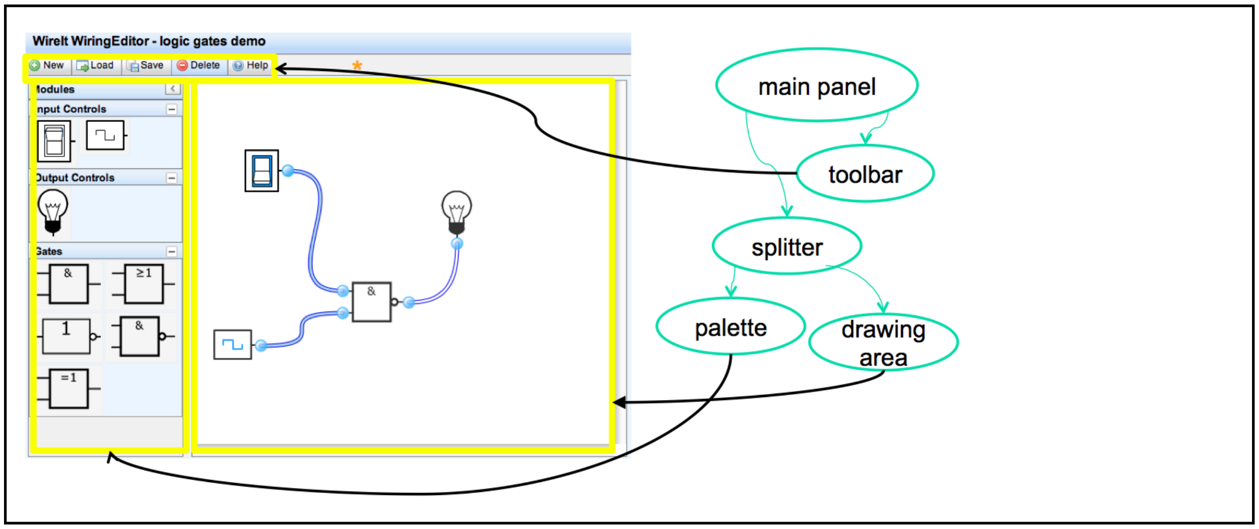
Today's reading is the first in a series of readings about how graphical user interfaces are implemented. Today we'll take a high-level look at the software architecture of GUI software, focusing on the **design patterns** that have proven most useful. Three of the most important patterns are the **model-view-controller** abstraction (not covered in this reading), 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, and JavaScript here. 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

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

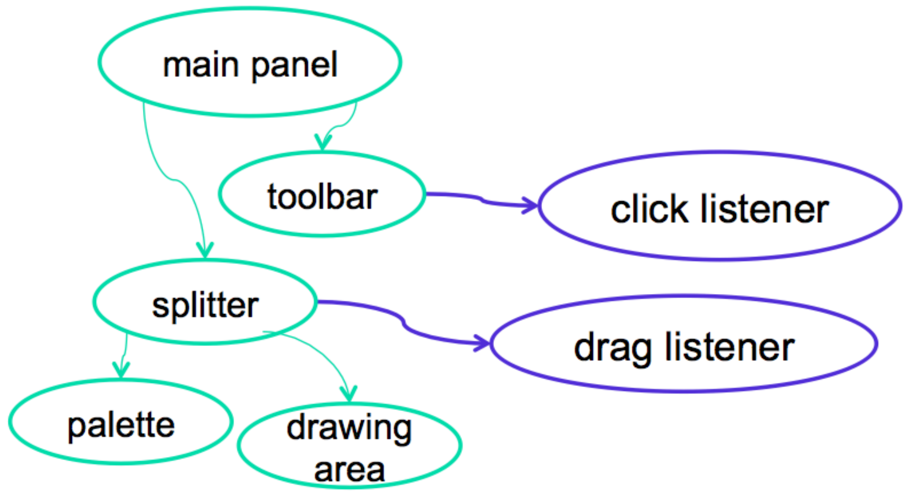
Virtually every GUI system has some kind of **view tree**. The view tree is a powerful structuring idea, which is loaded with responsibilities in a typical GUI:

- **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. A layout algorithm automatically calculates positions and sizes of views.

We'll look at more about each of these areas in later readings.

Input Handling

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



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

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.

GUI Implementation Approaches

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

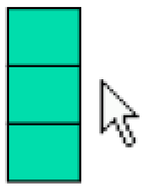
Procedural

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



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.

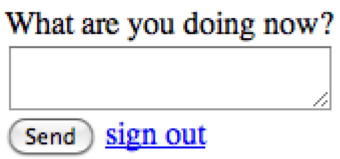
Finally, in **direct manipulation** style, the programmer uses a direct-manipulation graphical user interface to create the view tree. These interfaces are usually called GUI builders, and they offer a palette of view object classes, a drawing area to arrange them on, and a property editor for changing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

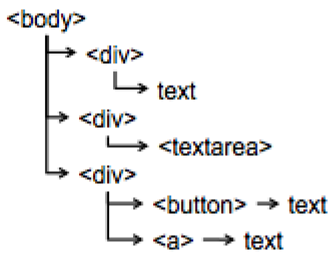
All three paradigms have their uses, but the sweet spot for GUI programming basically lies in an appropriate mix of declarative and procedural–which is what HTML/JavaScript provides.

Markup Languages

- HTML **declaratively** specifies a view tree

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





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.

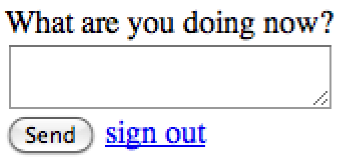
View Tree Manipulation

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

JavaScript
```javascript
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);
```



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, a set of parts and interactions between them, that allows an HTML document to be treated as an object in an object-oriented programming language.

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 use for the programming assignments.

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

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 a 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.

Revision Notes:

- Removed: MVC
- Removed: Declarative UI for Model/View Binding