Reading 24: Graphical User Interfaces

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

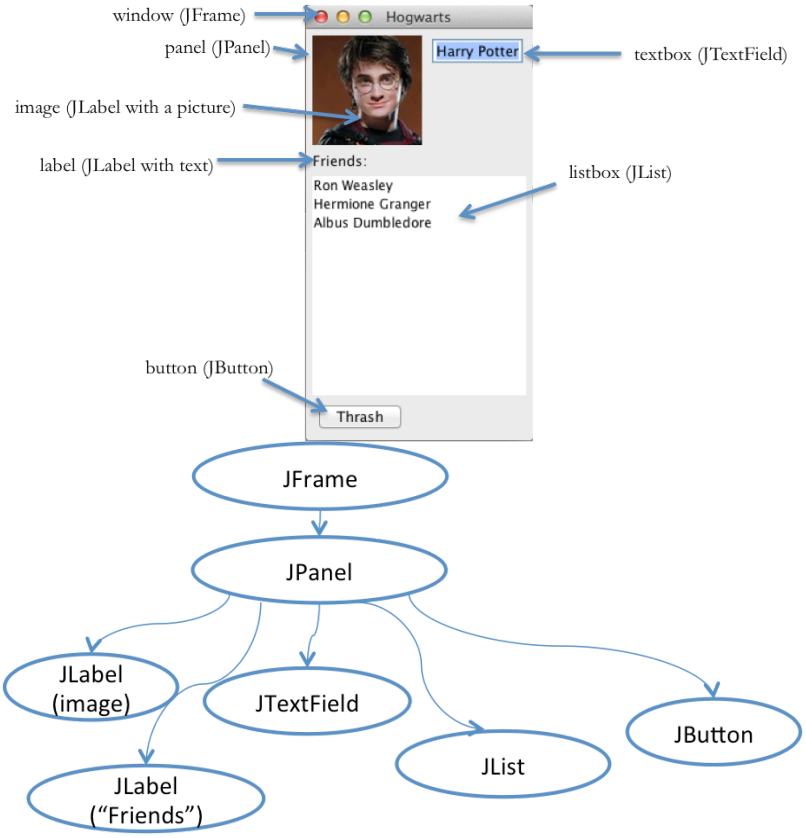
Today we'll take a high-level look at the software architecture of GUI software, focusing on the design patterns that have proven most useful. Three of the most important patterns are:

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

View Tree

Graphical user interfaces are composed of view objects, each of which occupies a certain portion 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 (https://en.wikipedia.org/wiki/Swing_(Java)) , they're `JComponent` objects; in HTML, they're elements or nodes; in other toolkits, they may be called widgets, controls, or interactors.

This leads to the first important pattern we'll talk about today: the view tree. 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

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 tree directs the display process. GUIs change their output by mutating the view tree. For example, to show a new set of photos in a photo album GUI, the current thumbnails are removed from the view tree and a new set of thumbnails is added in their place. A redraw algorithm built into the GUI toolkit automatically redraws the affected parts of the subtree. In Java Swing, every view in the tree has a `paint()` method that knows how to draw itself on the screen. The repaint process is driven by calling `paint()` on the root of the tree, which recursively calls `paint()` down through all the descendent nodes of the view tree.

Input . Views can have input handlers, and the view tree controls how mouse and keyboard input is processed. More on this in a moment.

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. Specialized containers (like `JSplitPane` , `JScrollPane`) do layout themselves. More generic containers (`JPanel` , `JFrame`) delegate layout decisions to a layout manager (e.g. `GroupLayout` , `BorderLayout` , `BoxLayout` , ...).

Input Handling

Input is handled somewhat differently in GUIs than we've been handling it in parsers and servers. In those systems, we've seen a single parser that peels apart the input and decides how to direct it to different modules of the program. If a GUI were written that way, it might look like this (in pseudocode):

```
while (true) {
    read mouse click
    if (clicked on Thrash button) doThrash();
    else if (clicked on textbox) doPlaceCursor();
    else if (clicked on a name in the listbox) doSelectItem();
    ...
}
```

In a GUI, we don't directly write this kind of method, because it's not modular – it mixes up responsibilities for button, listbox, and textbox all in one place. Instead, GUIs exploit the spatial separation provided by the view tree to provide functional separation as well. Mouse clicks and keyboard events are distributed around the view tree, depending on where they occur.

GUI input event handling is an instance of the **Listener pattern** (also known as 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 methods.

The control flow through a graphical user interface proceeds like this:

- A top-level event loop reads input from mouse and keyboard. In Java Swing, and most graphical user interface toolkits, this loop is actually hidden from you. It's buried inside the toolkit, and listeners appear to be called magically.
- For each input event, it finds the right view in the tree (by looking at the x,y position of the mouse) and sends the event to that view's listeners.
- Each listener does its thing (which might involve e.g. modifying objects in the view tree), and then *returns immediately to the event loop* .

The last part – listeners return to the event loop as fast as possible – is very important, because it preserves the responsiveness of the user interface. We'll come back to this later in the reading.

The Listener pattern isn't just used for low-level input events like mouse clicks and keyboard keypresses. Many GUI objects generate their own higher-level events, often as a result of some combination of low-level input events. For example:

- `JButton` sends an action event when it is pressed (whether by mouse or keyboard)
- `JList` sends a selection event when the selected element changes (whether by mouse or by keyboard)
- `JTextField` sends change events when the text inside it changes for any reason

A button can be pressed either by the mouse (with a mouse down and mouse up event) or by the keyboard (which is important for people who can't use a mouse, like blind users). So you should always listen for these high-level events, not the low-level input events. Use an `ActionListener` to respond to a `JButton` press, not a mouse listener.

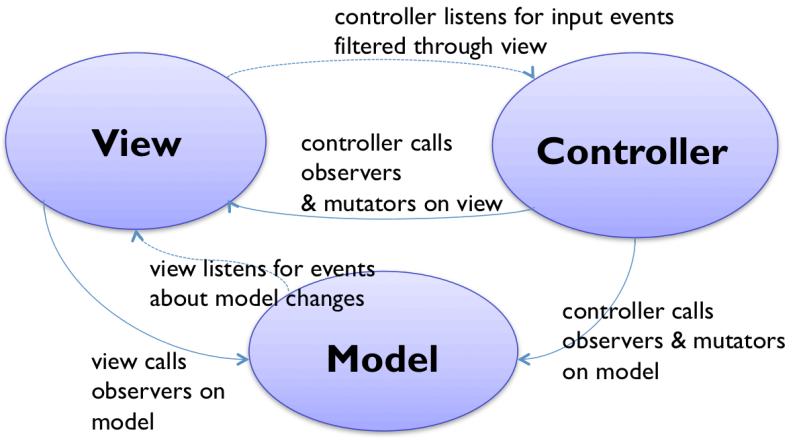
Separating Frontend from Backend

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 represents the data and logic that the user interface is showing and editing. (Why do we want to separate this from the user interface?)

The **Model-View-Controller pattern** has this separation of concerns as its primary goal. 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.

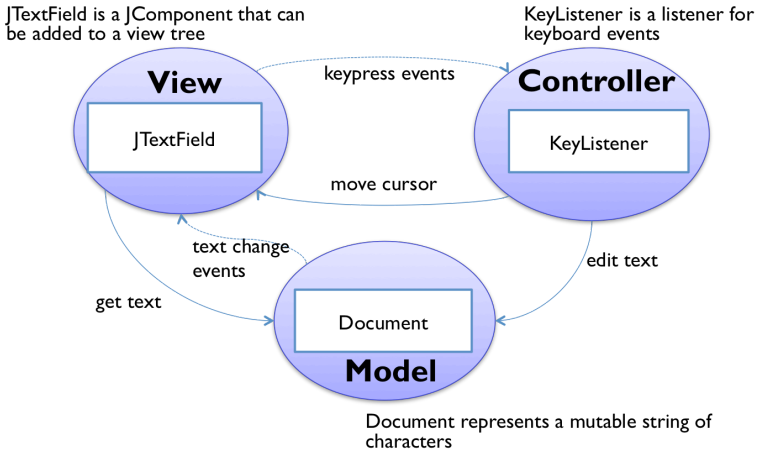
The **model** is responsible for maintaining application-specific data and providing access to that data. Models are often mutable, and they provide methods for changing the state safely, preserving its representation invariants. OK, all mutable objects do that. But a model must also notify its clients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

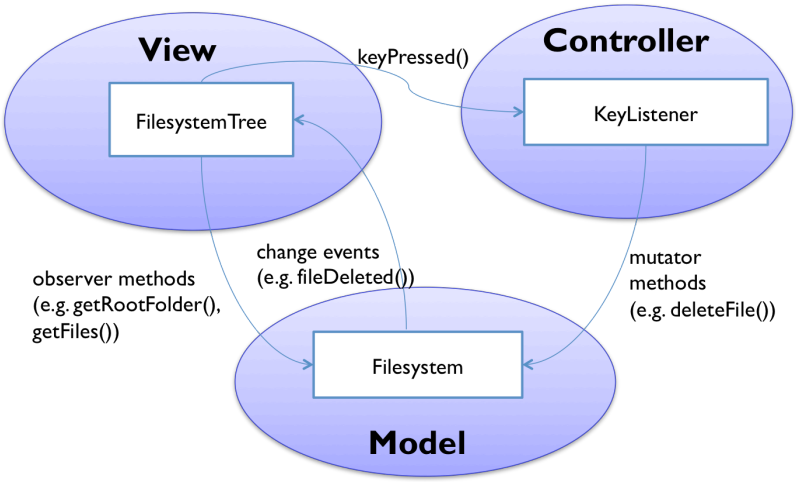
Finally, the **controller** handles the input. It receives keyboard and mouse events, and instructs the model to change accordingly.

A simple example of the MVC pattern is a text field. The figure at right shows Java Swing's text field, called `JTextField`. 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 into the mutable 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 an 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.



The separation of model and view 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 views. Java Swing is such a toolkit. You can easily reuse view classes from this library (like `JButton` and `JTree`) while plugging your own models into them.

Background Processing in Graphical User Interfaces

The last major topic for today connects back to concurrency.

First, some motivation. Why do we need to do background processing in graphical user interfaces? Even though computer systems are steadily getting faster, we're also asking them to do more. Many programs need to do operations that may take some time: retrieving URLs over the network, running database queries, scanning a filesystem, doing complex calculations, etc.

But graphical user interfaces are event-driven programs, which means (generally speaking) everything is triggered by an input event handler. For example, in a web browser, clicking a hyperlink starts loading a new web page. But if the click handler is written so that it actually retrieves the web page itself, then the web browser will be very painful to use. Why? Because its interface will appear to freeze up until the click handler finishes retrieving the web page and returns to the event loop. Here's why.

This happens because input handling and screen repainting is all handled from a single thread. That thread (called the event-dispatch thread) has a loop that reads an input event from the queue and dispatches it to listeners on the view tree. When there are no input events left to process, it repaints the screen. But if an input handler you've written delays returning to this loop – because it's blocking on a network read, or because it's searching for the solution to a big Sudoku puzzle – then input events stop being handled, and the screen stops updating. So long tasks need to run in the background.

In Java, the event-dispatch thread is distinct from the main thread of the program (see below). It is started automatically when a user interface object is created. As a result, every Java GUI program is automatically multithreaded. Many programmers don't notice, because the main thread typically doesn't do much in a GUI program – it starts creation of the view, and then the main thread just exits, leaving only the event-dispatch thread to do the main work of the program.

The fact that Swing programs are multithreaded by default creates risks. There's very often a shared mutable datatype in your GUI: the model. If you use background threads to modify the model without blocking the event-dispatch thread, then you have to make sure your data structure is threadsafe.

But another important shared mutable datatype in your GUI is the view tree. Java Swing's view tree is not threadsafe. In general, you cannot safely call methods on a Swing object from anywhere but the event-dispatch thread.

The view tree is a big meatball of shared state, and the Swing specification doesn't guarantee that there's any lock protecting it. Instead the view tree is **confined to the event-dispatch thread** , by specification. So it's ok to access view objects from the event-dispatch thread (i.e., in response to input events), but the Swing specification forbids touching – reading or writing – any `JComponent` objects from a different thread. See Swing threading and the event-dispatch thread (//www.javaworld.com/javaworld/jw-08-2007/jw-08-swingthreading.html) .

In the actual Swing implementation, there is one big lock (`Component.getTreeLock()` (https://docs.oracle.com/javase/8/docs/api/java/awt/Component.html#getTreeLock--)) but only some Swing methods use it, so it's not effective as a synchronization mechanism.

The safe way to access the view tree is to do it from the event-dispatch thread. So Swing takes a clever approach: it uses the event queue itself as a message-passing queue. In other words, you can put your own custom messages on the event queue, the same queue used for mouse clicks, keypresses, button action events, and so forth. Your custom message is actually a piece of executable code, an object that implements `Runnable` , and you put it on the queue using `SwingUtilities.invokeLater` (https://docs.oracle.com/javase/8/docs/api/javax/swing/SwingUtilities.html#invokeLater-java.lang.Runnable-) . For example:

```
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        content.add(thumbnail);
        ...
    }
});
```

The `invokeLater()` drops this `Runnable` object at the end of the queue, and when Swing's event loop reaches it, it simply calls `run()` . Thus the body of `run()` ends up run by the event-dispatch thread, where it can safely call observers and mutators on the view tree.

In the Java Tutorials, read:

- **Concurrency in Swing (//docs.oracle.com/javase/tutorial/uiswing/concurrency/index.html)** (1 page)
- **Initial Threads (//docs.oracle.com/javase/tutorial/uiswing/concurrency/initial.html)** (1 page)
- **The Event Dispatch Thread (//docs.oracle.com/javase/tutorial/uiswing/concurrency/dispatch.html)** (1 page)

Summary

- The view tree organizes the screen into a tree of nested rectangles, and it is used in dispatching input events as well as displaying output.
- The Listener pattern sends a stream of events (like mouse or keyboard events, or button action events) to registered listeners.
- The Model-View-Controller pattern separates responsibilities: model=data, view=output, controller=input.

- Long-running processing should be moved to a background thread, but the Swing view tree is confined to the event-dispatch thread. So accessing Swing objects from another thread requires using the event loop as a message-passing queue, to get back to the event-dispatch thread.