
CSE 143 Java

Models and Views

Reading: Ch. 18

1/18/2005

(c) 2001-5, University of Washington

07-1

Overview

- Topics
 - Displaying dynamic data
 - Model-View-Controller (MVC)
 - Observer Pattern

1/18/2005

(c) 2001-5, University of Washington

07-2

Review: Repainting the Screen

- GUI components such as JPanels can draw on a Graphics context by overriding `paintComponent`
- Problem: Drawings aren't permanent – need to be refreshed
 - Window may get hidden, moved, minimized, etc.
- Even components like buttons, listboxes, file choosers etc. also must render themselves
 - Seldom a reason to override paint methods for such components.
 - There are indirect but more convenient ways to change the rendering – e.g., changing the text of a label

1/18/2005

(c) 2001-5, University of Washington

07-3

Review: Using *paintComponent*

- Every Swing Component subclass has a *paintComponent* method
 - Called *automatically* by the system when component needs redrawing
- Program can override *paintComponent* to get access to the Graphics object and draw whatever is desired
- To request the image be updated, send it a `repaint()` message
 - `paintComponent()` is *eventually* called
- "Render" is the word for producing the actual visual image
 - Rendering may take place at multiple levels
 - Ultimate rendering is done by low-level software and/or hardware

1/18/2005

(c) 2001-5, University of Washington

07-4

Drawing Based on Stored Data

- Problem: how does `paintComponent()` know what to paint?
 - What is painted might change over time
- Answer: we need to store the information somewhere
- Where?
 - Store detailed graphical information in the component
 - Lines, shapes, colors, positions, etc.
 - Probably in an instance variable, accessible to `paintComponent`
 - Store *underlying* information in the component
 - Store objects that know how to paint themselves
 - Store references to the underlying data and query it as needed
 - data object returns information in a form that might differ from the underlying data
 - `paintComponent` translates the data into graphics
- All of these approaches *can* be made to work. What is best?

1/18/2005

(c) 2001-5, University of Washington

07-5

Model-View-Controller Pattern

- Idea: want to separate the underlying data from the code that renders it
 - Good design because it separates issues, reduces coupling
 - Allows multiple views of the same data
- Model-View-Controller pattern
 - Originated in the Smalltalk community in 1970's
 - Used throughout Swing
 - Although not always obvious on the surface
 - Widely used in commercial programming
 - Recommended practice for graphical applications

1/18/2005

(c) 2001-5, University of Washington

07-6

MVC Overview

- **Model**
 - Contains the "truth" – data or state of the system
- **View**
 - Renders the information in the model to make it visible to users in desired formats
 - Graphical display, dancing bar graphs, printed output, network stream....
- **Controller**
 - Reacts to user input (mouse, keyboard) and other events
 - Coordinates the models and views

1/18/2005

(c) 2001-5, University of Washington

07-7

MVC Interactions and Roles (1)

- **Model**
 - Maintains the data in some internal representation
 - Maintains a list of interested viewers
 - Notify viewers when model has changed and view update might be needed
 - Supplies data to viewers when requested
 - Possibly in a different representation
 - Generally should not know about display or user interface details

1/18/2005

(c) 2001-5, University of Washington

07-8

MVC Interactions and Roles (2)

- *View*
 - Maintains details about the display environment
 - Gets data from the model when it needs to
 - Renders data when requested (by the system or the controller, etc.; in Java, often implements `paintComponent` to do this)
 - May catch user interface events and notify controller
- *Controller*
 - Intercepts and interprets user interface events
 - Routes information to models and views

1/18/2005

(c) 2001-5, University of Washington

07-9

MVC vs MV

- Separating Model from View...
 - ...is just good, basic object-oriented design
 - usually not hard to achieve, with forethought
- Separating the Controller from the View is a bit less clear-cut
- Often the Controller and the View are naturally closely related – buttons or mouse clicks on a panel in a `JFrame`, for instance
 - Controller and view frequently use GUI Components
 - OK to fold view and controller together *when it makes sense*
 - Fairly common in modern user interface packages

1/18/2005

(c) 2001-5, University of Washington

07-10

Implementation Note

- Model, View, and Controller are design concepts, not class names
- Might be more than one class involved in each
- Can have multiple views and controllers (only 1 model)
- The View might involve a number of different GUI components
- MVC might apply at multiple levels in a system
 - A Controller might use a listbox to interact with a user.
 - That listbox is part of the Controller
 - However, the listbox *itself* has a Model and a View, and possibly a Controller

1/18/2005

(c) 2001-5, University of Washington

07-11

Observer Pattern

- The MVC design is a particular instance of a more general idea: the “observer” pattern
- Key idea: object that might change keeps a list of interested observers and notifies them when something happens
 - Observers can react however they like
- Support in the Java library: interface `java.util.Observer` and class `java.util.Observable`
 - Model implements `Observer`
 - Observers register themselves with `Observable` objects and are notified when they change
 - Use this if you want, but can be overkill for simple projects
 - CSE143 demo programs do this by hand for clarity

1/18/2005

(c) 2001-5, University of Washington

07-12