CSE 143 Java

Models and Views

Reading: Ch. 20

Overview

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

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* for such components. There are indirect but more convenient ways to change the rendering

Review: Using *paintComponent*

- Every JComponent subclass has a *paintComponent* method
  - Called *automatically* by the system when component needs redrawing
  - In AWT, use *paint* instead
- Program can override *paintComponent* to get access to the Graphics object and draw what 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

Drawing Based on Stored Data

- Problem: how does paintComponent( ) know what to paint?
  - This might need to change over time, too
- Answer: we need to store the information somewhere
- Where? Four ideas:
  - 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?

Model-View-Controller Pattern

- Idea: want to *separate* the underlying data from the code that renders it
  - Good design because it separates issues
  - Consistent with object-oriented principles
  - 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

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
    Might create the model or view
    Might pass a model reference to a view or vice versa
  - Sometimes a test method or main method acts as a controller

MVC Example: Chess Game

- The model knows the state of the game
  - what pieces are where
  - whose turn it is
  - what the rules of moving and playing are
- The view displays the game state
  - might display a standard chess board with symbols
  - might display a fanciful board with human figures
  - might list all the moves made so far, with no board!
- The controller makes things happen
  - sets up the game initially
  - lets the players make their moves
  - reports illegal moves, won/loss status, time left, etc.

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 be unaware of the display or user interface details

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.)
  - May catch user interface events and notify controller
- *Controller*
  - Intercepts and interprets user interface events
  - Routes information to models and views

M-V-C vs M-V

- 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
  - May be overkill in a small system
- Often the Controller and the View are naturally closely related – buttons on a panel in a JFrame, for instance
  - Both frequently use GUI Components
  - OK to fold view and controller together *when it makes sense*

Implementation Notes

- Model, View, and Controller are design concepts, not class names
- Might be more than one class involved in each.
- The View might involve a number of different GUI components
  - displaying the state of the model
- The Controller might also involve GUI components
  - interacting with the user
- 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.

Observer Pattern

- The MVC design is a particular use of a more general "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: class java.util.Observable and interface java.util.Observer
  - A model might extend Observable
    In practice, the Observable class per se is not always used
  - Observers register themselves with Observable objects and are notified when they change