CSC 143

Models and Views

Reading: Ch. 18

Overview

• Topics
  • Displaying dynamic data
  • Model-View
  • Model-View-Controller (MVC)
• Reading:
  • Textbook: Ch. 20

Review: Repainting the Screen

• GUI components such as JPanels can draw on themselves using a Graphics context
• 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.
• Solution: A “callback” method called paintComponent

Review: Using paintComponent

• Or just plain paint for older AWT components.
  • Every Component subclass has a paint (paintComponent) method
  • Called automatically by the system when component needs redrawing
  • Program can override paintComponent to get the Graphics and draw what is desired
  • To request the image be updated, send it a “repaint” message
  • paintComponent() is eventually called
  • Footnote: "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?
  • The picture might need to change over time, too.
• Answer: we need to store the information somewhere
• Where? Some possibilities
  • 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
  - "Model" is a poor word. "Content" or "underlying data" would be better.
- **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

MVC Interactions and Roles

- **Model**
  - Maintains the data in some internal representation
  - Supplies data to view when requested
    - Possibly in a different representation
  - Advanced: Notifies viewers when model has changed and view update might be needed
    - Generally unaware of the display details
- **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.)
  - Advanced: Catches user interface events and notifies controller
- **Controller**
  - Intercepts and interprets user interface events
  - Routes information to models and views

MVC vs MV

- Separating Model from View...
  - ...is just good, basic object-oriented design
  - Usually not hard to achieve, with forethought
  - Separating the Controller is a bit less clear at
    - May be overlapped in a small system.
  - Often the Controller and the View are naturally closely related
    - Both frequently use GUI Components, which the Model is unlikely to do.
- Model View Pattern: **MV**
  - Folds the Controller and the View together.

Implementation Note

- 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
  - Example: JFileChooser
- 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.

Example: Simple Simulator Framework

- Class SimModel – model for a particle simulation
  - (Same basic idea as uwcsesim, but simpler)
  - SimModel maintains the state of the simulation – keeps track of the objects that have been added to the world
- Interface SimThing – anything that implements this can be added to the simulation
- Interface SimView – anything that implements this can be a viewer of the model
- (No controller for this example)

Model-View Interaction

- It's possible to have more than one viewer
- A viewer tells the model that it wants to be notified when something interesting happens
- The model contains a list of all interested viewers
- When something happens (a cycle in the simulation has occurred, for example), the model calls the notify() method of each viewer
  - Viewers can react however they like
  - This illustrates the "observer pattern"
  - Used heavily in the Java user interface libraries, among other places
An Example Simulation

• Class Ball implements SimThing
  • A bouncing ball that updates its position on each action() and
    reverses direction if it hits the edge
  • Implements paintComponent(Graphics g) to draw itself when asked
• Class BallGraphicsView implements SimView
  • A JPanel that is notified after each cycle of the simulation
    just requests repaint()
  • Method paintComponent gets the list of all Ball objects from the
    model and asks each one to paint itself using the supplied Graphics
    object