Model-View-Controller (MVC) Design Pattern

Overview

- Displaying Dynamic Data
- Model-View-Controller (MVC) Pattern
- Observer Pattern
- Delegate-Model 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 paintComponent methods for such components.

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

Drawing Based on Stored Data

- Problem: how does paintComponent( ) know what to paint?
  - What is painted might change over time, too
- 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?

MVC Motivation (1)

- 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

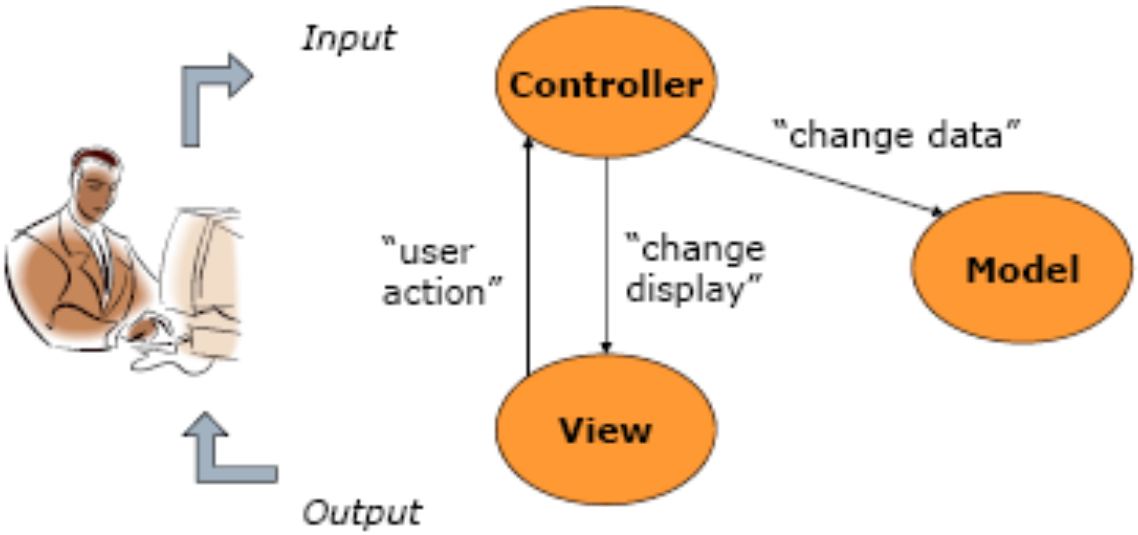
MVC Motivation (2)

- Basic parts of any application:
  - Data being manipulated
  - A user-interface through which this manipulation occurs
- The data is logically independent from how it is displayed to the user
  - Display should be separately designable/evolvable
- Example: grade distribution in class
  - Displayed as both pie chart and/or bar chart
- Anti-example: see BigBlob
  - Presentation, logic, and state all mixed together

Model-View-Controller Pattern

- Model
  - The data (i.e. state)
  - Methods for accessing and modifying state
- View
  - Renders contents of model for user
  - When model changes, view must be updated
- Controller
  - Translates user actions (i.e. interactions with view) into operations on the model
  - Example user actions: button clicks, menu selections

Basic Interactions in MVC



Implementing Basic MVC in Swing

- Mapping of classes to MVC parts
  - View is a Swing widget (like a JFrame & JButton)
  - Controller is an ActionListener
  - Model is an ordinary Java class (or database)
- Alternative mapping
  - View is a Swing widget and includes (inner) ActionListener(s) as event handlers
  - Controller is an ordinary Java class with “business logic”, invoked by event handlers in view
  - Model is an ordinary Java class (or database)
- Difference: Where is the ActionListener?
  - Regardless, model and view are completely decoupled (linked only by controller)

Mechanics of Basic MVC

- Setup
  - Instantiate model
  - Instantiate view
    - Has reference to a controller, initially null
  - Instantiate controller with references to both
    - Controller registers with view, so view now has a (nonnull) reference to controller
- Execution
  - View recognizes event
  - View calls appropriate method on controller
  - Controller accesses model, possibly updating it
  - If model has been changed, view is updated (via the controller)

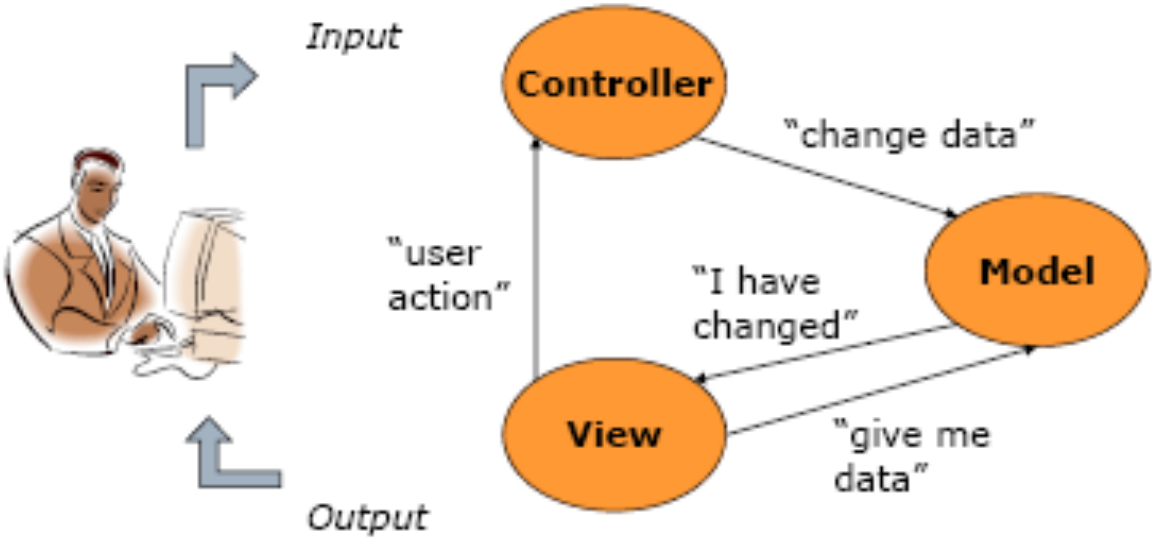
Mechanics of Basic MVC

- Example: CalcMVC
  - CalcModel, CalcView, CalcController

Problems with Classic MVC

- Controller might need to produce its own output
  - e.g. Popup menu
- Some state is shared between controller and view, but does not belong in model
  - e.g. Selection (highlighted text)
- Direct manipulation means that user can interact (control) visual elements (views)
  - e.g. Scrollbar
- Overall issue: Input and output are often intermingled in a GUI
  - Result: View and controller are tightly coupled

Extended Interactions in MVC



Role of Extended Pattern

- Background: 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.Observer and interface java.util.Observable
  - Model implements Observable
  - Observers register themselves with Observable objects and are notified when they change
- In extended MVC, view is an observer of model

Role of Extended Pattern

Application within MVC

- Asynchronous model updates
  - Model changes independent of user actions
  - Associated view must be notified of change in order to know that it must update
- A model may have multiple views
  - But a view has one model
  - All views have to be updated when model changes

Model

- Represents application data and business rules that govern access to and updates of this data
  - In enterprise software, a model often serves as a software approximation of a real-world process
- Maintains a list of interested viewers
- Notifies views when it changes and enables the view to query
- Allows the controller to access application functionality encapsulated by the Model

Model Tasks

- Store and manage data elements, such as state information
- Respond to queries about its state
- Respond to instructions to change its state
- e.g., the model for a radio button can be queried to determine if the button is pressed
- Generally should not know details of the display or user interface details

View

- Renders the contents of a model
- Specifies how the model data should be presented
- When the model changes, the view must update its presentation
  - push model
    - the view registers itself with the model for change notifications
  - pull model
    - the view is responsible for calling the model when it needs to retrieve the most current data
- Maintains details about the display environment

View Tasks

- Implements a visual display of the model
- e.g., a button has a colored background, appears in a raised perspective, and contains an icon and text; the text is rendered in a certain font in a certain color
- Forwards user gestures to the controller

Controller

- Defines application behavior
- Interprets user gestures and maps them into actions
  - For the model to perform
  - In selecting a different view
    - e.g., a web page of results to present back to the user
- In a Web application, user gestures appear as HTTP requests

Controller Tasks

- Receive user inputs from mouse and keyboard
- Map these into commands that are sent to the model and/or view to effect changes in the view
- e.g., detect that a button has been pressed and inform the model that the button stated has changed

Mechanics of Extended MVC (1)

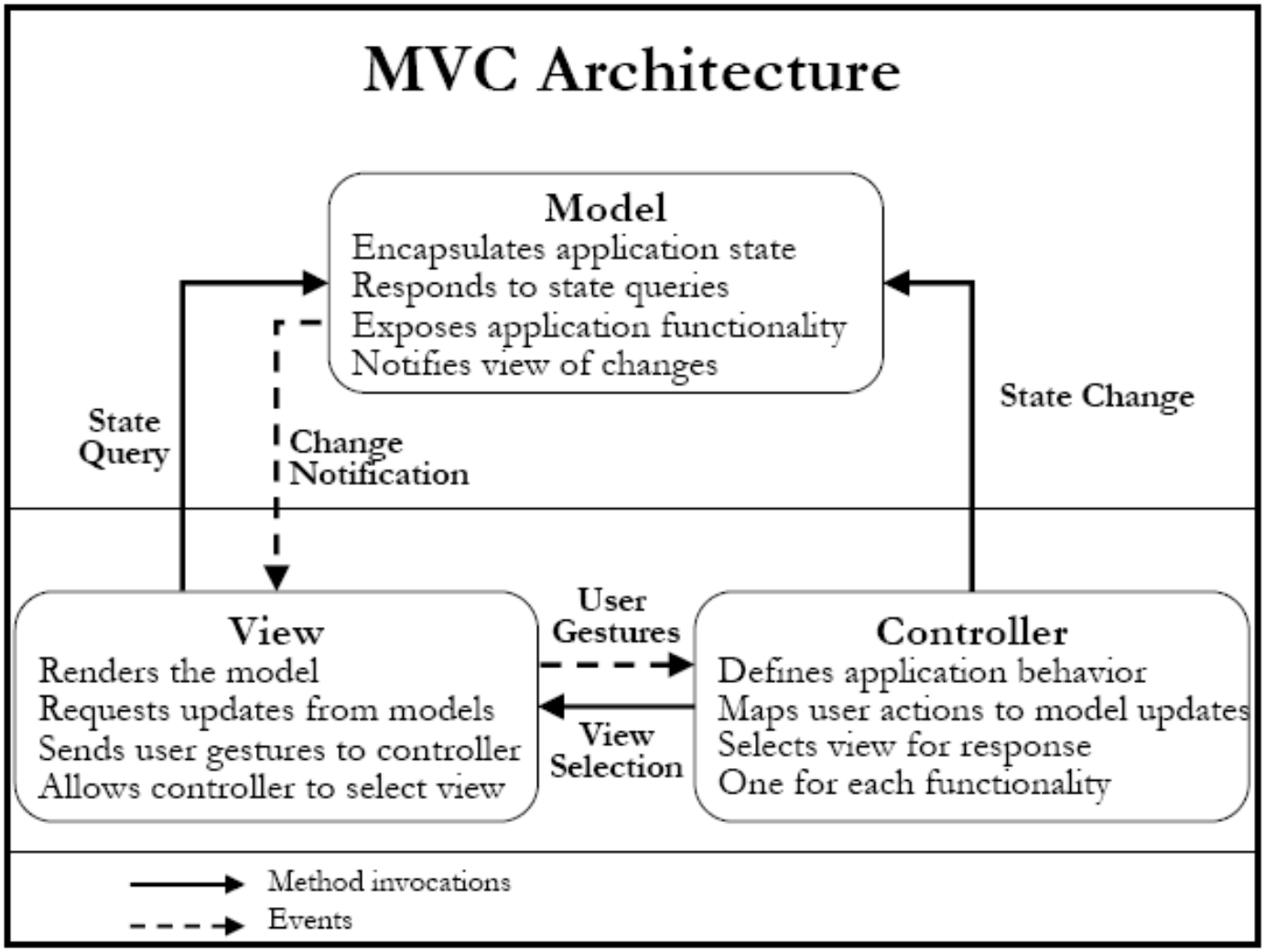
Setup

- Instantiate model
  - Has reference to view, initially null
- Instantiate view with reference to model
  - View registers with model
- Instantiate controller with references to both
  - Controller registers with view

Mechanics of Extended MVC (2)

Execution

- View recognizes event
- View calls appropriate method on controller
- Controller accesses model, possibly updating it
- If model has been changed, it notifies all registered views
- Views then query model for the nature of the change, rendering new information as appropriate


MVC Architecture
Model
Encapsulates application state
Responds to state queries
Exposes application functionality
Notifies view of changes
State Query
Change Notification
State Change
View
Renders the model
Requests updates from models
Sends user gestures to controller
Allows controller to select view
User Gestures
View Selection
Controller
Defines application behavior
Maps user actions to model updates
Selects view for response
One for each functionality
Method invocations
Events

Interaction between MVC Components (1)

Once the model, view, and controller objects are instantiated, the following occurs:

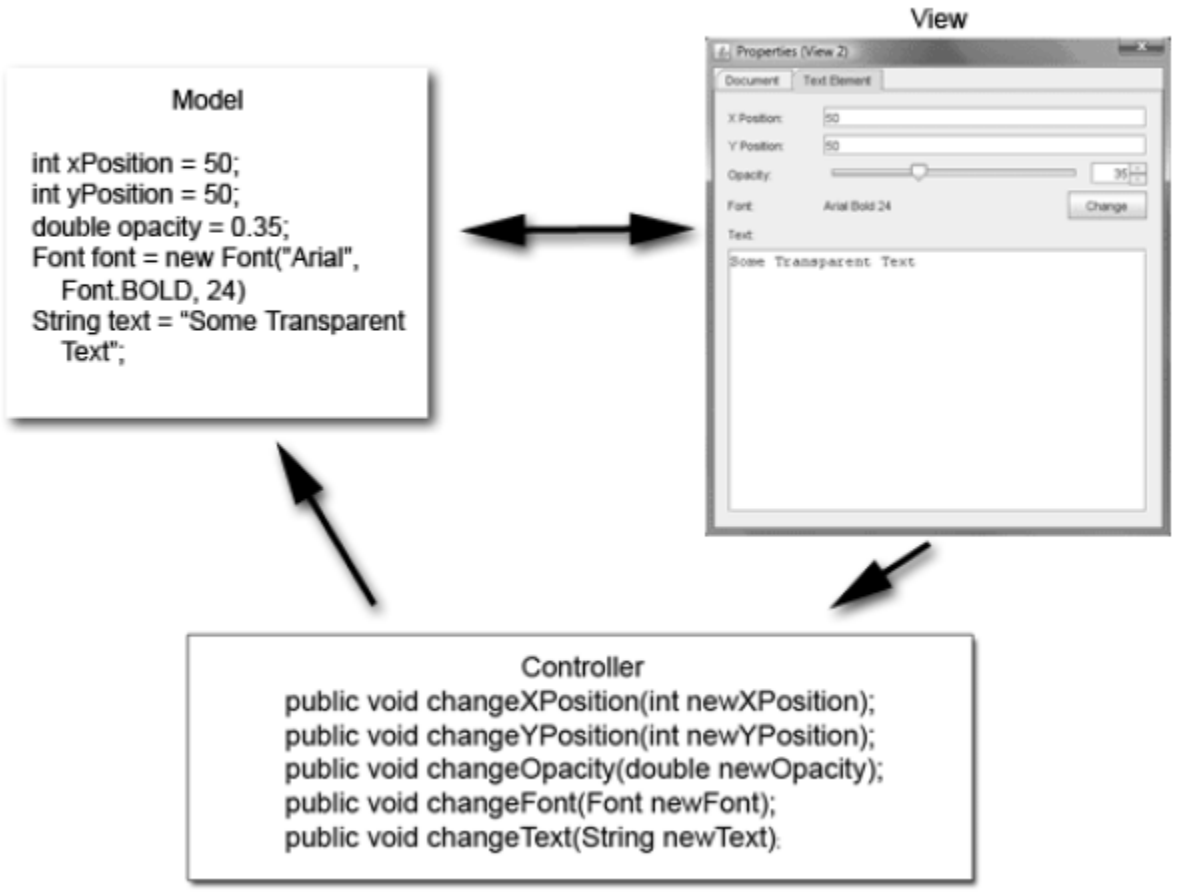
1. The view registers as a listener on the model
   - Any changes to the underlying data of the model immediately result in a broadcast change notification, which the view receives
   - This is an example of the push model described earlier
   - The model is not aware of the view or the controller
     - It simply broadcasts change notifications to all interested listeners
2. The controller is bound to the view
   - i.e., any user actions that are performed on the view will implicitly invoke a registered event listener method in the controller class
3. The controller is given a reference to the underlying model

Interaction between MVC Components (2)

Once a user interacts with the view, the following actions occur:

1. The view recognizes that a user action has occurred
2. The view generates an event, which implicitly invokes appropriate method in the controller
3. The controller accesses the model
   - Possibly updating it with respect to the user's action
4. If the model has been altered, it notifies interested listeners, such as the view, of the change
   - The controller may also update the view

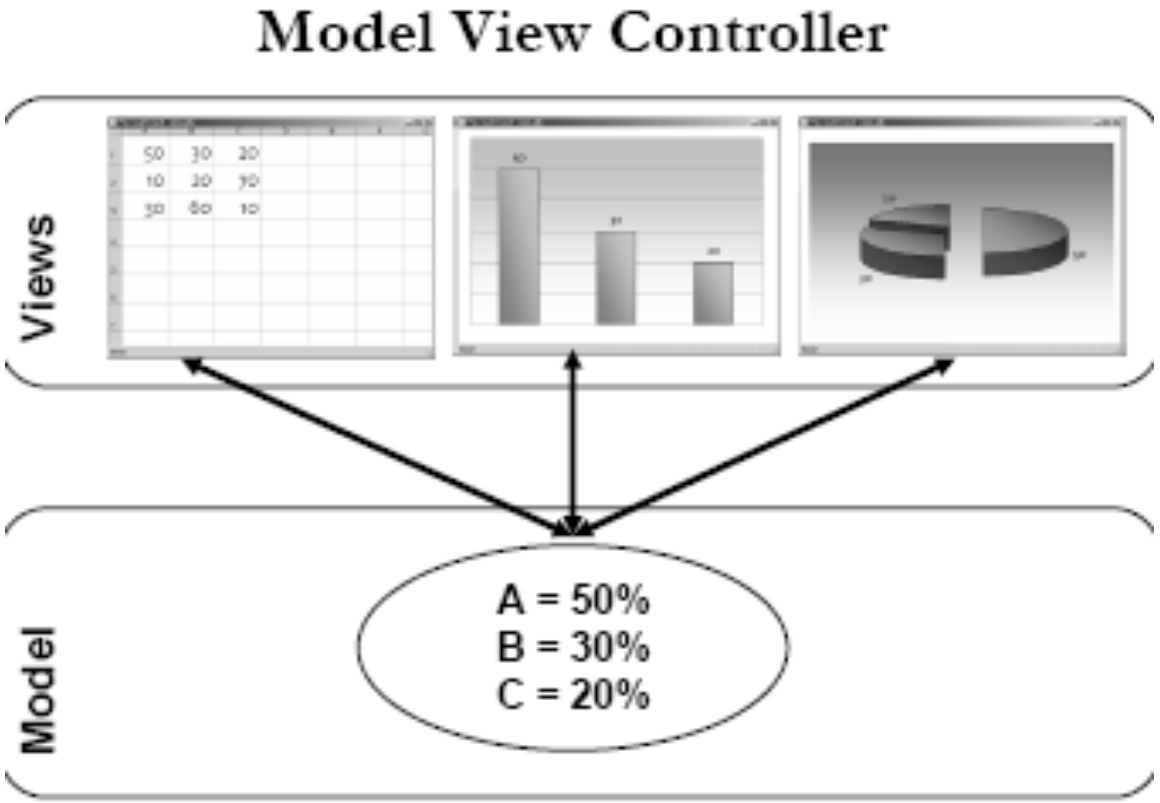
MVC Example


View
Model
int xPosition = 50;
int yPosition = 50;
double opacity = 0.35;
Font font = new Font("Arial",
Font.BOLD, 24)
String text = "Some Transparent
Text";
Properties (View 2)
Document
Text Element
X Position:
Y Position:
Opacity:
Font:
Arial Bold 24
Change
Text:
Some Transparent Text
Controller
public void changeXPosition(int newXPosition);
public void changeYPosition(int newYPosition);
public void changeOpacity(double newOpacity);
public void changeFont(Font newFont);
public void changeText(String newText);

Implementation Note (1)

- 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

Implementation Note (1)



Implementation Note (2)

- 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

Implementation Note (2)

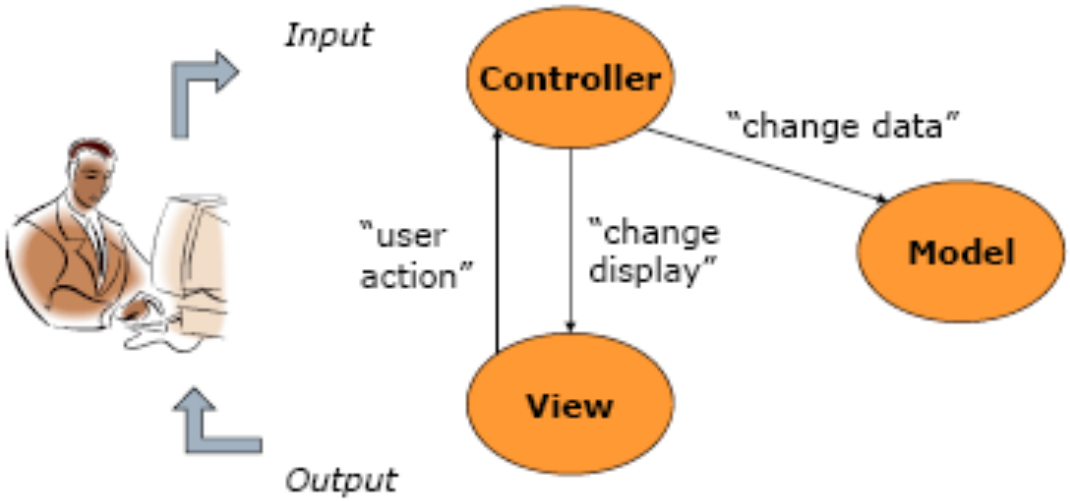
MVC vs. MV

- 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

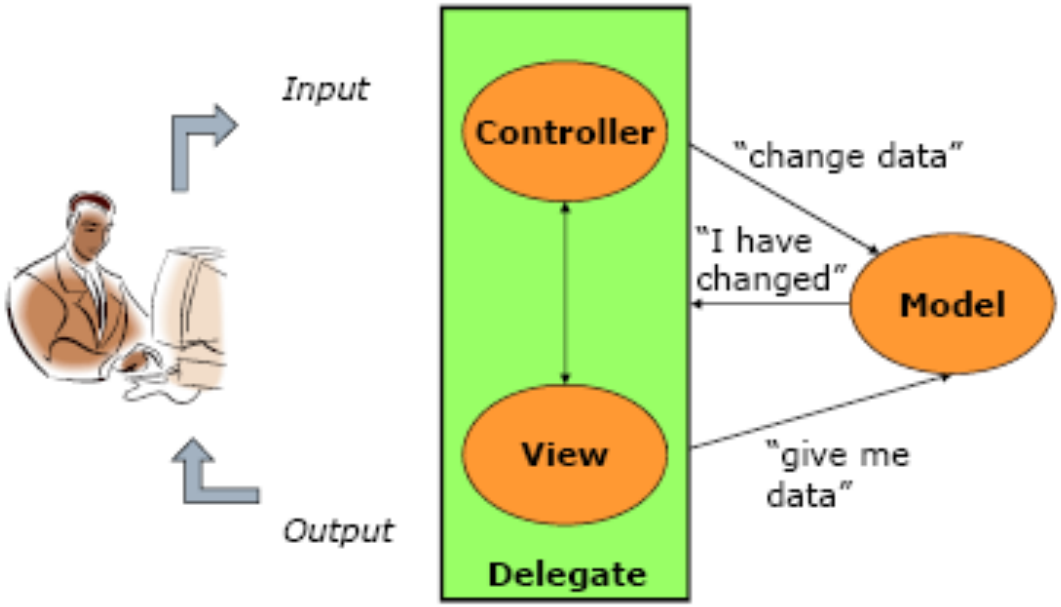
Delegate-Model Pattern

- Model
  - Data, same as before
- Delegate
  - Responsible for both input and output
  - A combination of both view and controller
- Many other names
  - UI-Model
  - Document-View

Basic Interactions in Delegate Model



Basic Interactions in Delegate Model



Mechanics of Delegate Model

- Setup
  - Instantiate model
  - As with MVC, model does not know/care about UI
  - Instantiate delegate with reference to model
- Execution
  - Delegate recognizes event and executes appropriate handler for the event
  - Delegate accesses model, possibly updating it
  - If model has been changed, UI is updated

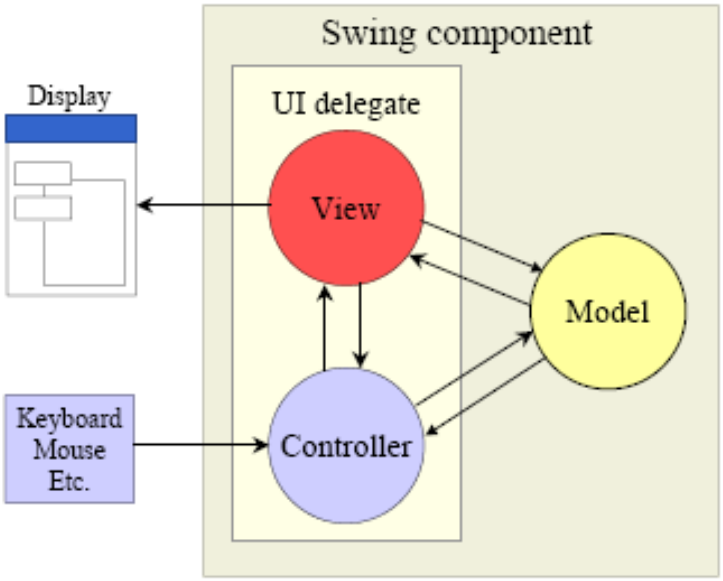
Mechanics of Delegate Model

- Example: CalcV3
  - CalcModel, CalcViewController
  - Note: CalcModel is exactly the same as with CalcMVC

MVC and Swing

- Swing designers found it difficult to write a generic controller that didn't know the specifics about the view
- So, they collapsed the view and controller into a single UI (user interface) object known as a delegate (the UI is delegated to this object)
- This object is known as a UI delegate

MVC and Swing



Notes

- Litmus test: Swapping out user interface
  - Can the model be used, without modification, by a completely different UI?
  - e.g. Swing vs console text interface
- Model can be easily tested with Junit
- Model actions should be quick
  - GUI is frozen while model executes
  - Alternative: multithreading, which gets much more complicated

Benefits of MVC Architecture

- Improved maintainability
  - Due to modularity of software components
- Promotes code reuse
  - Due to OO approach (e.g., subclassing, inheritance)
- Model independence
  - Designers can enhance and/or optimize model without changing the view or controller
- Plugable look and feel
  - New L&F without changing model
  - Multiple views use the same data
- Frameworks that emphasize MVC
  - Apache Struts, JSF, Ruby on Rails

Supplemental Reading

- http://java.sun.com/developer/technicalArticles/javase/mvc/
- http://onjava.com/onjava/2004/07/07/genericmvc.html
- http://www.cis.upenn.edu/~matuszek/cit591-2002/Examples/mvc.html
- http://www.csis.pace.edu/~bergin/mvc/mvcgui.html
- http://leepoint.net/notes-java/GUI/structure/40mvc.html

References

- P. Bucci, CSE 421, “Model-View-Controller (MVC) Design Pattern”, lecture25.pdf
- V. Offenback, CSE 143, “Models and Views”, 07-mvc.pdf
- W. Liu, CSC309F, “MVC (Model-View-Controller)”, mvc-rest-gdata.pdf
- S. MacKenzie, 3461A, “Model-View Controller: Advanced GUI concepts”, f03-a-08.pdf