Model-View Controller
IAT351

Week 17 Lecture 1

15.10.2012

Lyn Bartram

lyn@sfu.ca

SFU

SCHOOL OF INTERACTIVE ARTS + TECHNOLOGY

Administrivia

- CHANGE to assignments and grading
- 4 assignments
- This one (Assignment 3) is worth 20%
- Assignment 4 is worth 25% (it's a longer one)

- Final project proposals should be in to me by the beginning of next week.

Administrivia

- Assignment 3
- Multiple user interfaces to your same application
- The “regular” laptop/desktop view you have been assuming
- A mobile phone-size version

- 2 parts to this assignment:
  - DESIGN CHOICES
  - MODULAR CODE

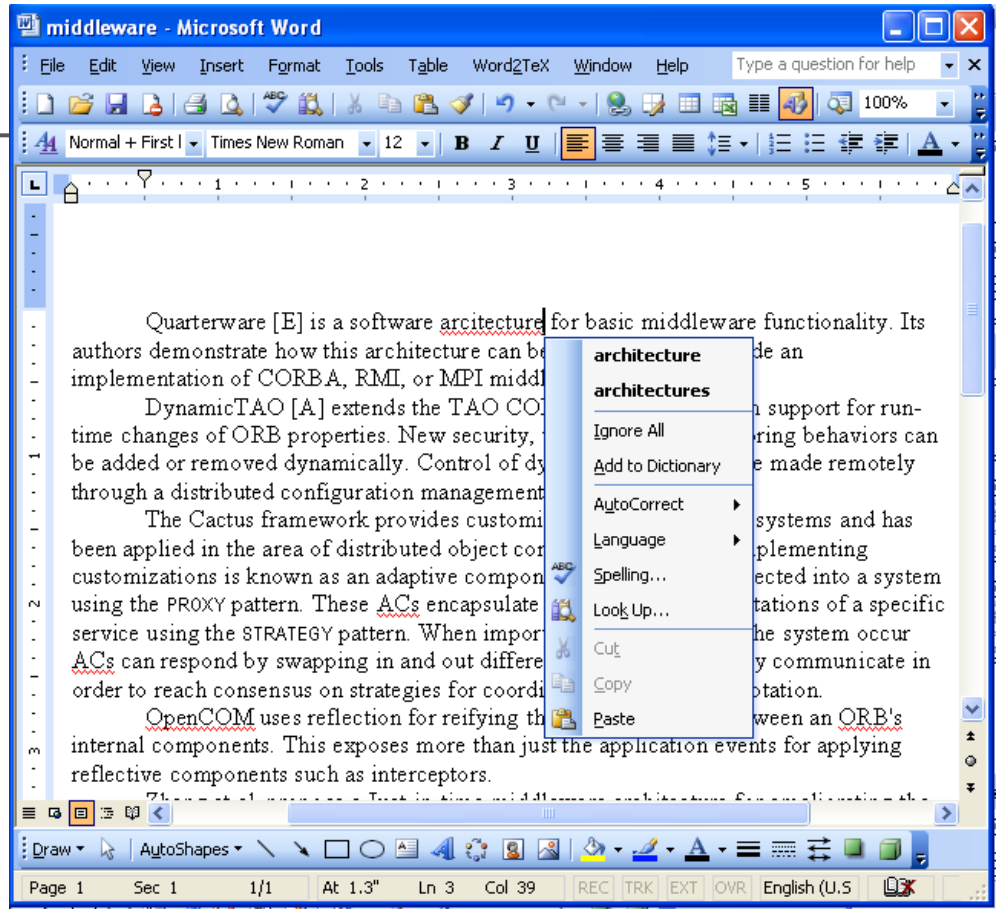
Problems with GUIs

User interfaces are especially prone to changes in requirements

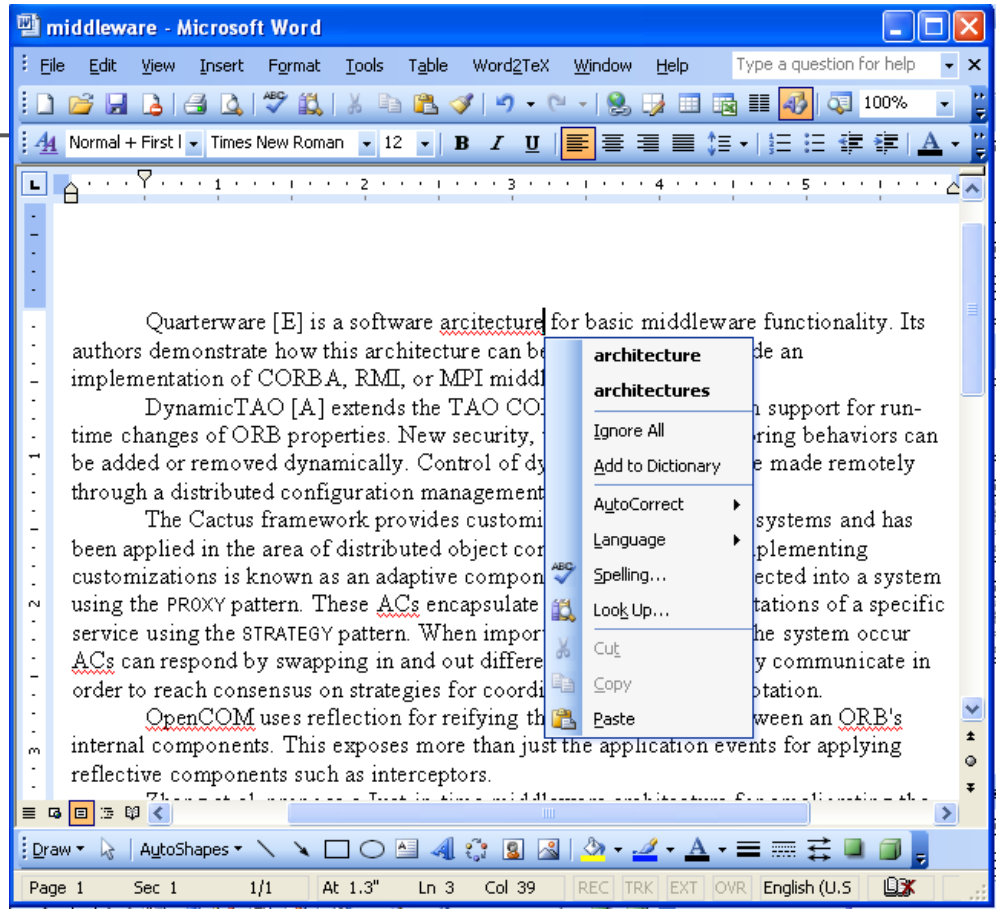
- New types of input
  - Keyboard
  - Mouse
  - Pen
  - Remote
- New types of output
  - Porting to different “look-and-feel”
  - Information visualization: charts, graphs, plots
  - Output device heterogeneity: phones, applets, Javascript, HTML, Swing

- New features require UI changes
  - ex. Add in-text spellcheck to word processor
- User must have a way to access the feature
- Feedback

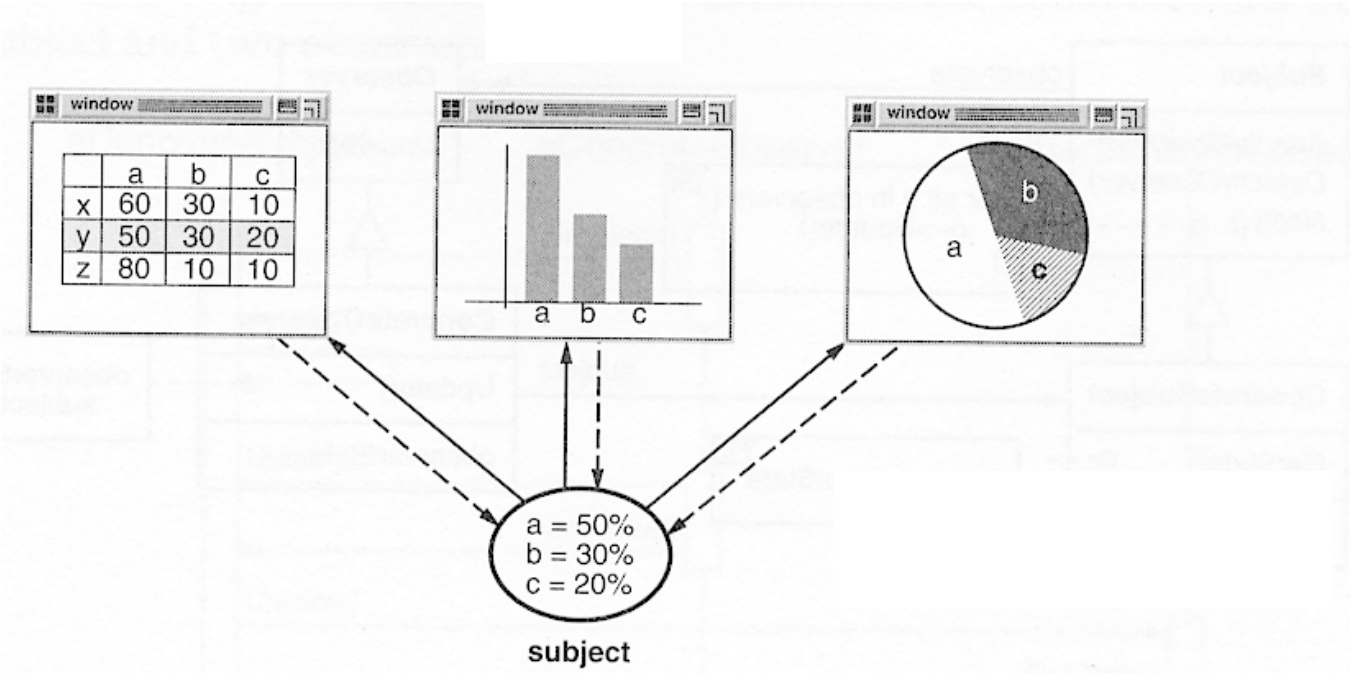
- What is changing "underneath the hood"?



- Changes to UI: Need to input corrections and update the underlying document *model*
- Changes to underlying document model: **None**



Multiple views of same computation



Separate the user interface from the application logic

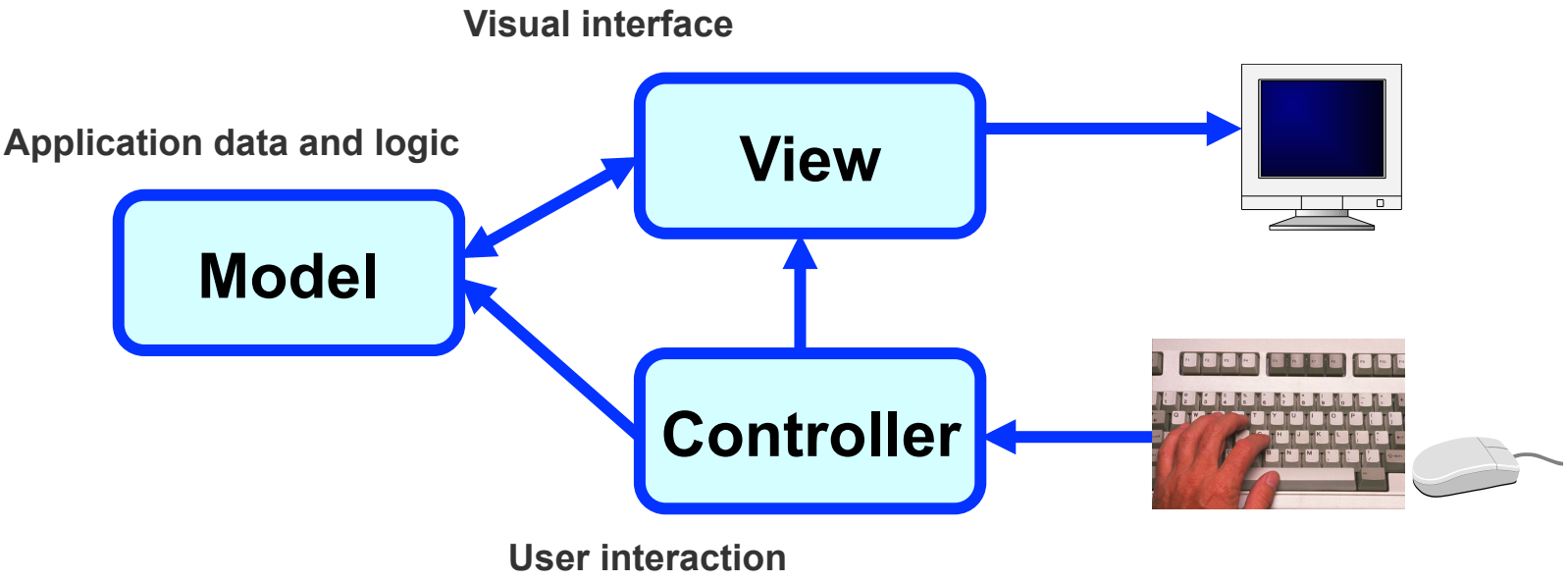
- Many different ways to present and interact with the same underlying information
- Presentation and interaction needs to be only loosely coupled to the underlying computational information abstraction
- How do we implement this?

MVC: Model-View-Controller

- Architectural pattern for building systems
- Divide system responsibilities into three parts
  - **Model** : Contains all program data and logic
  - **View** : Visual representation of model
  - **Controller** : manages input and system behavior
- Step by step
  - User uses **controller** to change data in model
  - **Model** then informs view(s) of change
  - **View** changes visual presentation to reflect change

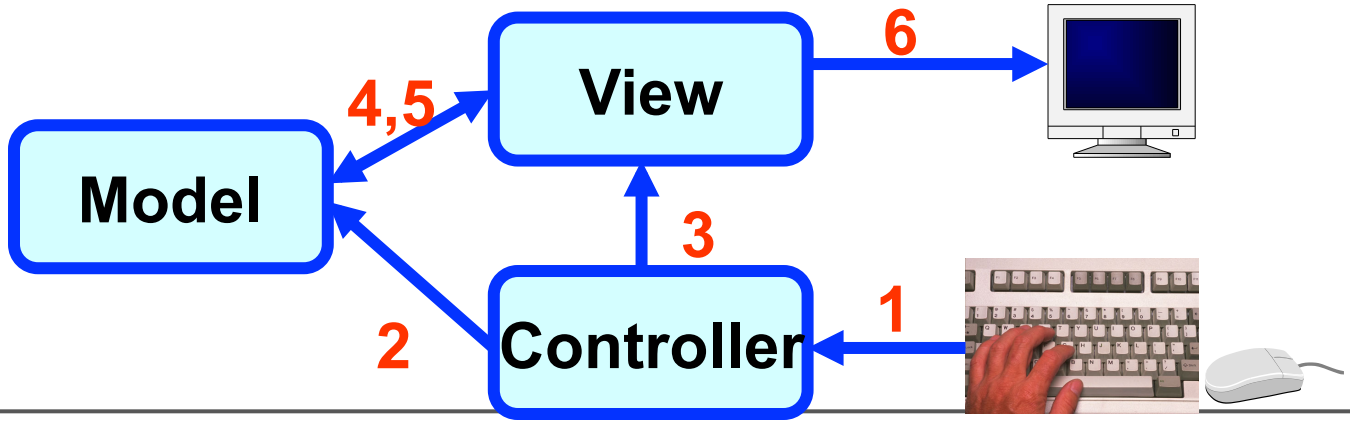
Model-View-Controller (MVC) Pattern

- Developed at Xerox PARC in 1978 for Smalltalk™



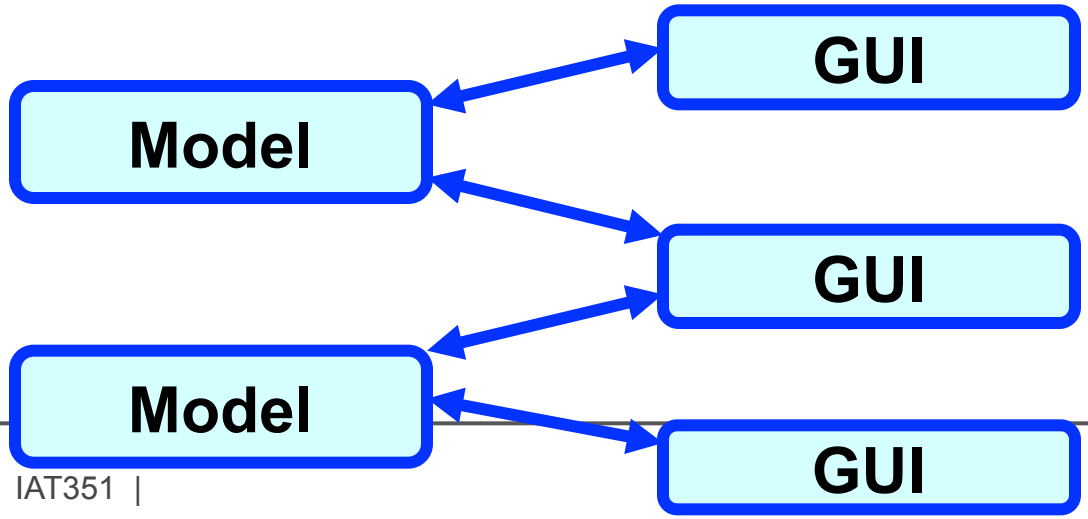
MVC Interaction Order

1 User performs action, controller is notified
2 Controller may request changes to model
3 Controller may tell view to update
4 Model may notify view if it has been modified
5 View may need to query model for current data
6 View updates display for user



MVC Pattern – Advantages

- Separates data from its appearance
  - More robust
  - Easier to maintain
- Provides control over interface
- Easy to support multiple displays for same data



MVC Pattern – Model

- Contains application & its data
  - Provide methods to access & update data
- Model interface defines allowed interactions
  - Fixed interface enable both model & GUIs to be easily pulled out and replaced
- Examples
  - Text documents (DOM – document object model)
  - Spreadsheets
    - provides a number of services to manipulate the data e.g., recalculate, save
    - computation and persistence issues
  - Web browser

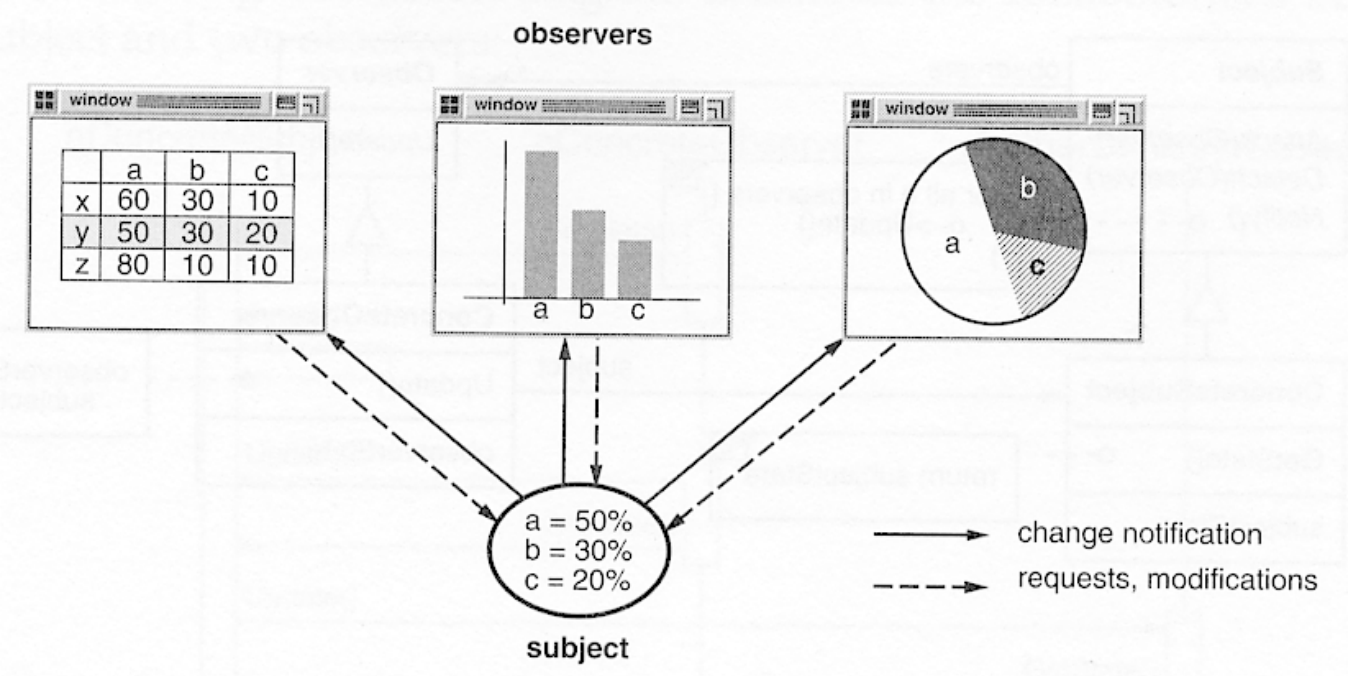
MVC Pattern – View

- Provides visual representation of model
- Multiple views can display model at same time
  - Example: data represented as table and graph
- When model is updated, all its views are informed & given chance to update themselves

MVC Pattern – Controller

- Users interact with the controller
- Interprets mouse movement, keystrokes, etc.
- Communicates those activities to the model
- Interaction with model causes view(s) to update

Dependencies



Principles of GUI Design

- Model
  - perform actual work (logic)
  - independent of the GUI ;provide access methods
- Controller
  - Lets user control what work the program is doing
  - Design of controller depends on model
- View
  - Lets user see what the program is doing
  - Should not display what controller thinks is happening (base display on model, not controller)

Principles of GUI Design

- Model is separate
  - Never mix model code with GUI code
  - View should represent model as it really is
    - Not some remembered status
- In GUIs, user code for view and controller tend to mingle
  - Especially in small programs
- To date, you largely have a mishmash of these functions in your application … except for ..

Do you have a good model?

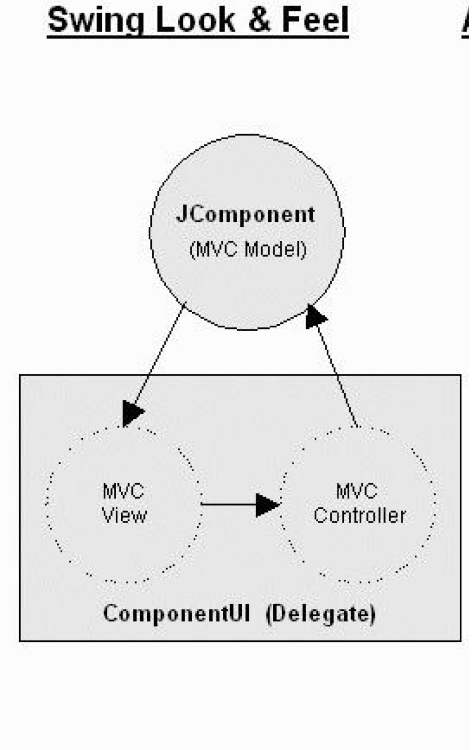
- Could you reuse the model if you wanted to port the application to:
  - a command-line textual interface
  - an interface for the blind
  - an iPod
  - a web application, run on the web server, accessed via a web browser

Dependencies

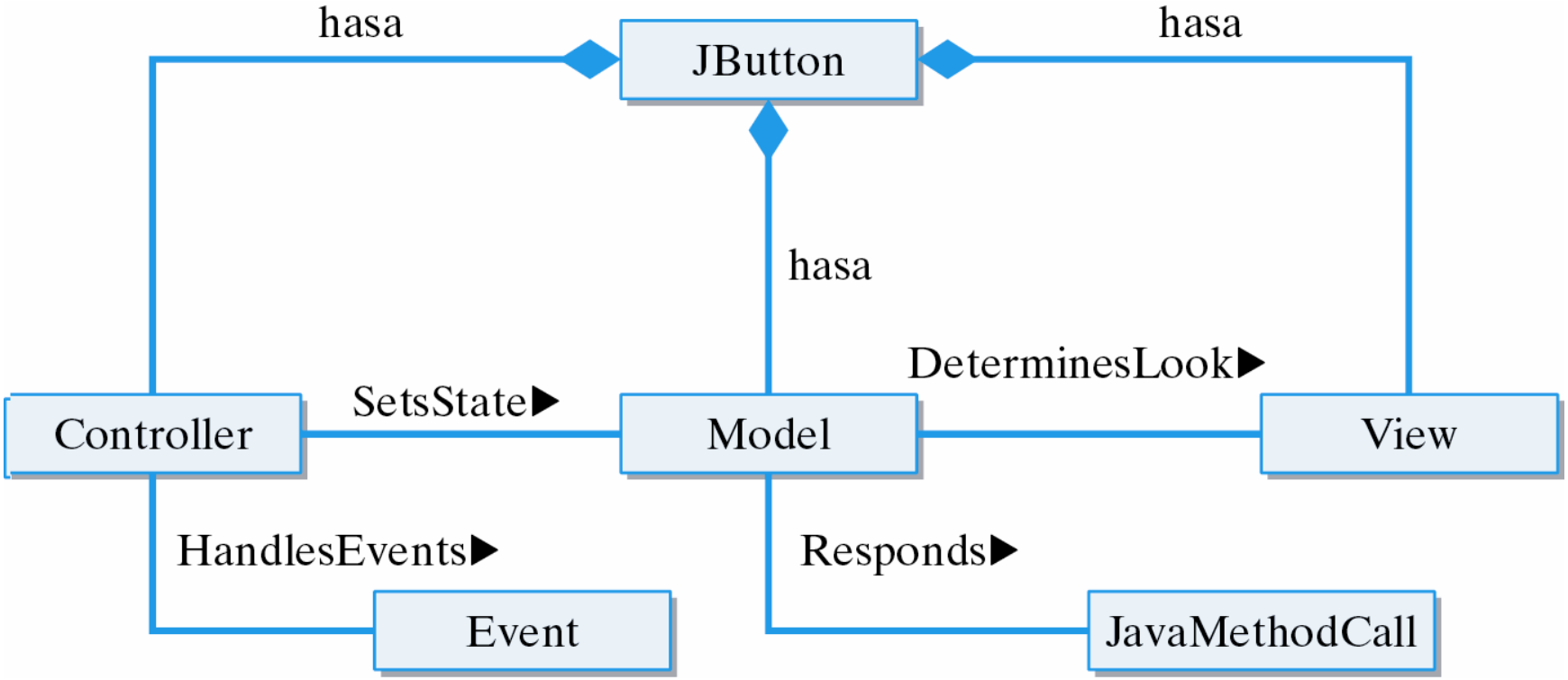
- Issues
  - need to maintain consistency in the views (or observers)
  - need to update multiple views of the common data model (or subject)
  - need clear, separate responsibilities for presentation (look), interaction (feel), computation, persistence

Swing components are internally based on MVC

- the component is divided into three separate objects:
  - *view*: how it looks (output/display)
  - *model*: what state it is in (data)
  - *controller*: what user input it accepts and what it does (input/events)



Internal MVC components within `JButton`



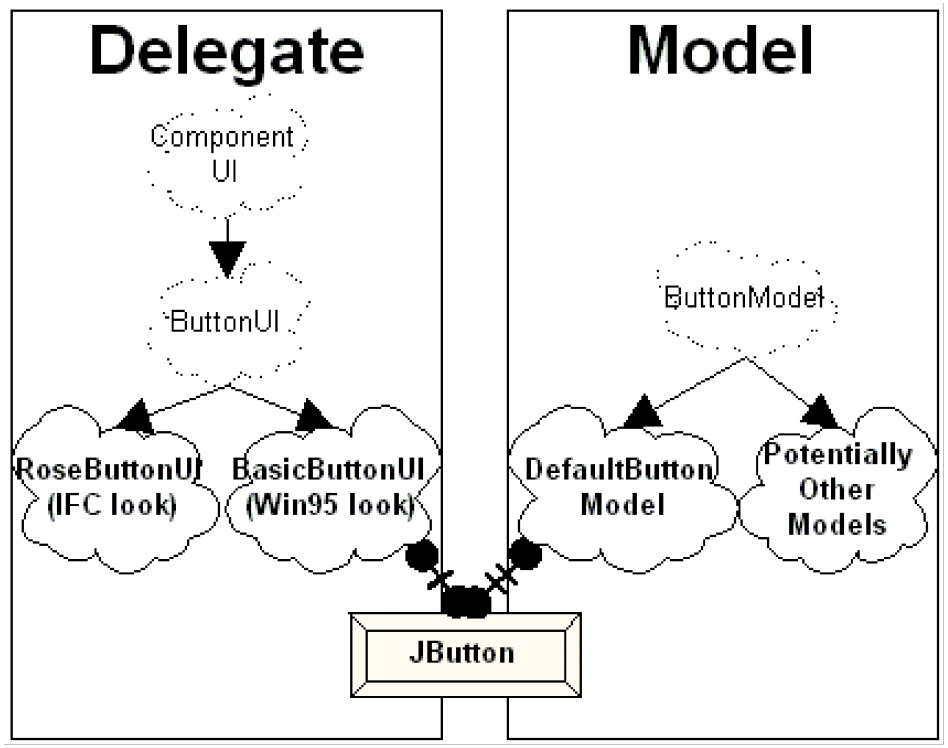
Fundamental principle

- Separation:
  - you can modify or create views without affecting the underlying model
  - the model should not need to know about all the kinds of views and interaction styles available for it

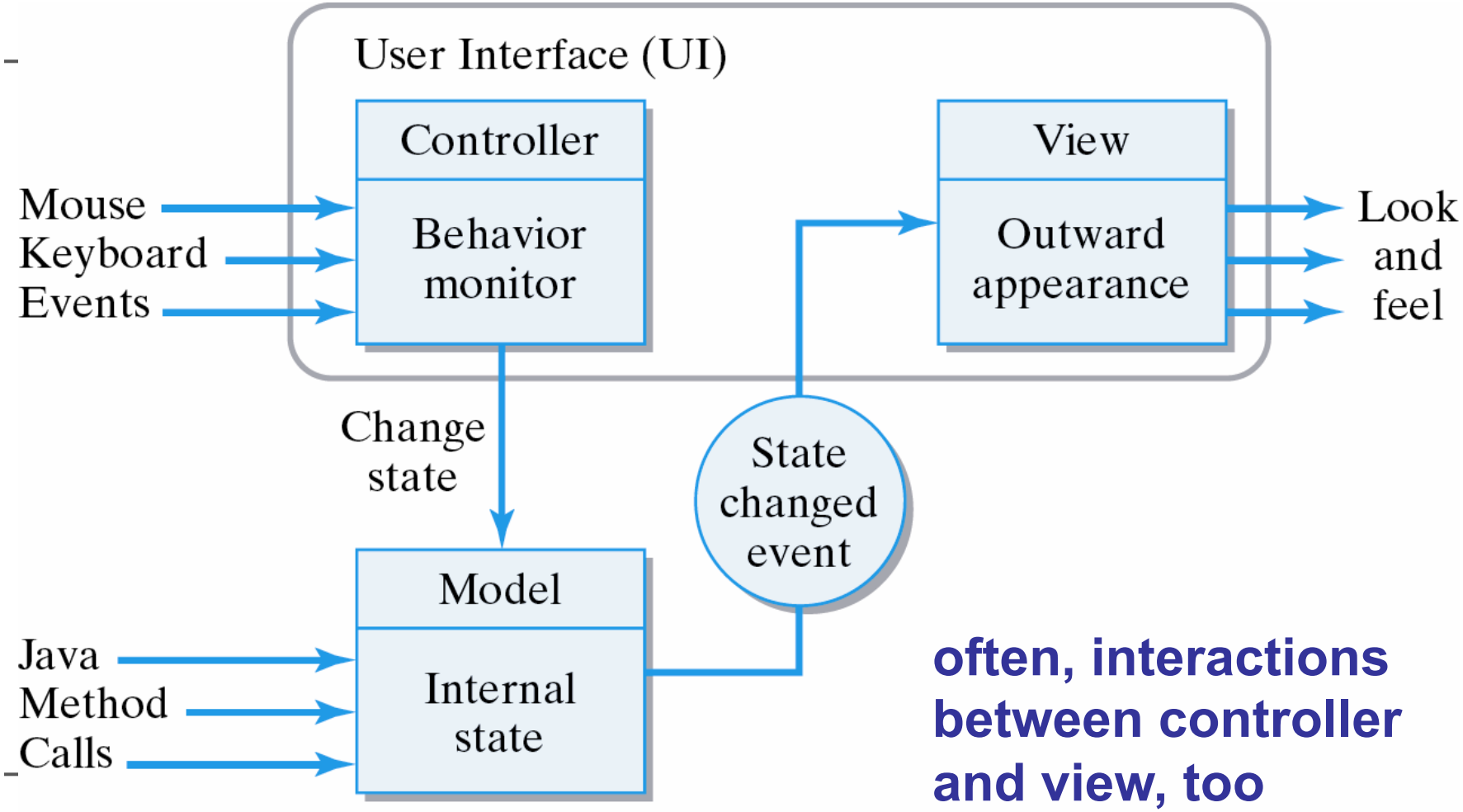
- How do we do this in Swing?
  - the programmer has the responsibility of program modularization
  - can put data into graphical components (bad style), or represent it separately (a better way)

Delegate model

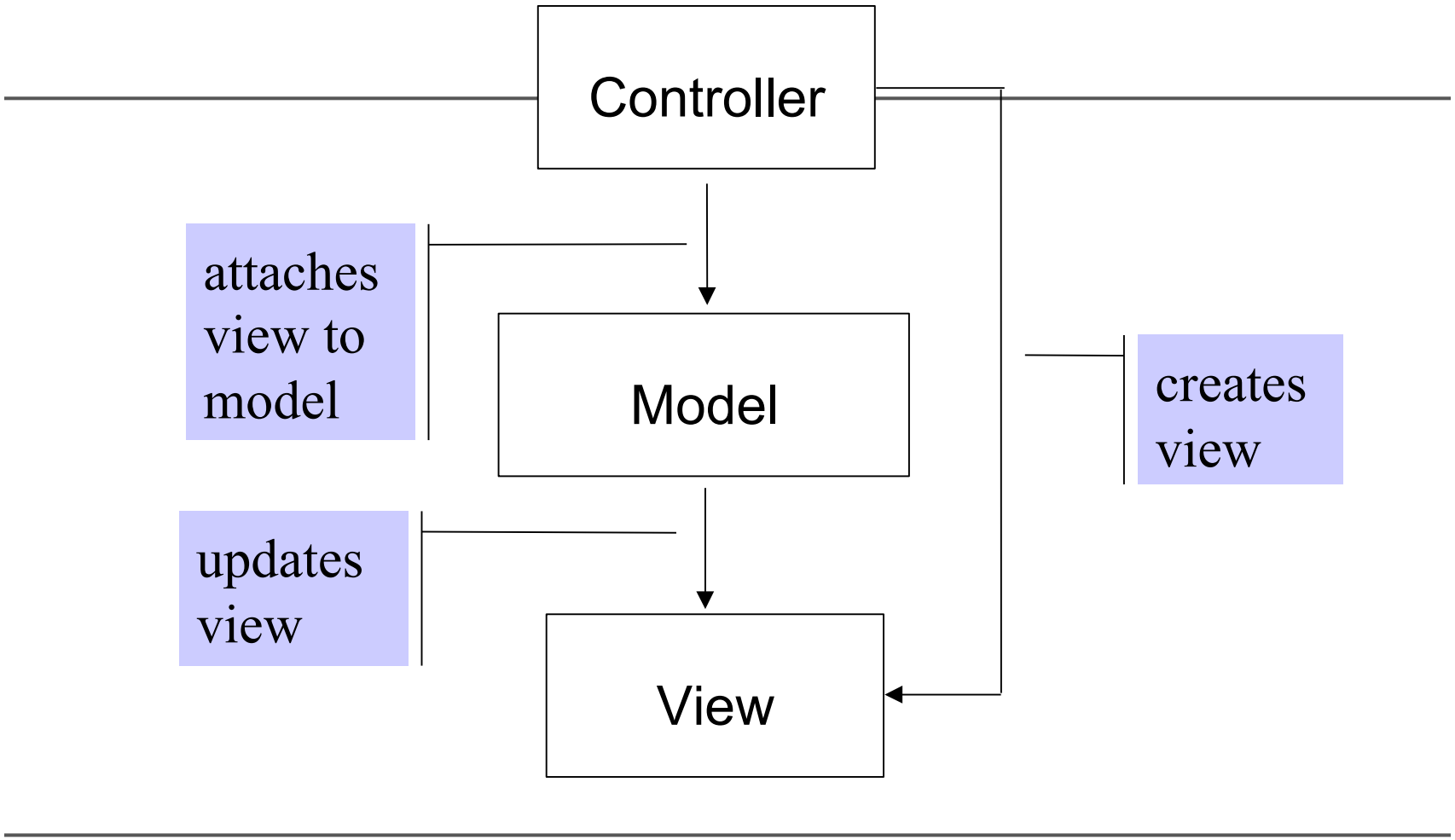
- In practice, Swing provides a *delegate* approach
- View and controller may share same "delegate class"
  - Inner class
- Controllers are basically listeners



Model-View-Controller (MVC)



often, interactions between controller and view, too


Controller
attaches view to model
Model
creates view
updates view
View

Java supports MVC with its `Observable` library class and `Observer` interface

- A "model" class extends `Observable`, which provides methods for attaching observers and notifying them when a change occurs
- A "view" class implements `Observer` and must supply the `update` method, called automatically when the model changes
- A "controller" class implements at least one `Listener` interface

- A "manager" class creates these and attaches/detaches them to each other.
- Often the controller class is the "manager", but this is entirely up to the individual application structure

Simple example

Demo (src is on the website)

4 classes:

- `JournalModel.java` **`extends java.util.Observable`**
- `JournalView.java` **`implements java.util.Observer`**
- `JournalController.java` **`implements ActionListener`**
- `RunJournalMVC.java`

JournalModel

```
public void setValue(int value)
{
    this.entryCounter = value;
    setChanged();
 notifyObservers(entryCounter);
} //setValue()

public void incrementValue()
{
    entryCounter++;
 setChanged();
 notifyObservers(entryCounter);
}
```

JournalView

```
public void update(Observable
obs, Object obj)
{
//who called us and what did
they send?
myTextField.setText("" +
((Integer)obj).intValue());
//obj is an Object, need to
cast to an Integer
}
```

JournalController Manager (RunVMC)

```
addModel(JournalModel m)
{
    this.model=m;
}

actionPerformed(ActionEvent e)
{
     model.incrementValue();
}
```

```
//create Model and View
myModel = new JournalModel();
myView1 = newJournalView();
bview = new
JournalView(400,400,200,200,
Color.YELLOW);

myModel.addObserver(myView1);
myModel.addObserver(bview);

myController = new
JournalController();
myController.addModel(myModel);
```

JournalController Manager (RunVMC)

```
addModel(JournalModel m)
{
    this.model=m;
}

actionPerformed(ActionEvent e)
{
     model.incrementValue();
}
```

```
//add controllers to view
because of BUTTON

myView1.addController(myController)
;
bview.addController(myController);
}
```

Summary: MVC Advantages

- Input processing is separated from output processing.
- Controllers can be interchanged, allowing different user interaction modes.
- Multiple views of the model can be supported easily.