Model and view

• **model**: Classes in your system that are related to the internal representation of the state and behavior of the system.
  - often part of the model is connected to file(s) or database(s)
  - examples (card game): Card, Deck, Player
  - examples (bank system): Account, User, UserList

• **view**: Classes in that display the state of the model to the user.
  - generally, this is your GUI (could also be a text UI)
  - should not contain crucial application data
  - Different views can represent the same data in different ways
    • Example: Bar chart vs. pie chart
  - examples: PokerGUI, PacManCanvas, BankApplet
**Model-view-controller**

- **model-view-controller (MVC)**: Design paradigm for graphical systems that promotes strict separation between model and view.

- **controller**: classes that connect model and view
  - defines how user interface reacts to user input (events)
  - receives messages from view (where events come from)
  - sends messages to model (tells what data to display)
Model/view separation

• Your model classes should NOT:
  ▪ import graphical packages (java.awt.*, javax.swing.*)
  ▪ store direct references to GUI classes or components
  ▪ know about the graphical classes in your system
  ▪ store images, or names of image files, to be drawn
  ▪ drive the overall execution of your program

• Your view/controller classes should:
  ▪ store references to the model class(es)
  ▪ call methods on the model to update it when events occur

• *Tricky part:* Updating all aspects of the view properly when the state of the model changes...
Pattern: Observer

objects that listen for updates to the state of others
Observer pattern

- **observer**: An object that "watches" the state of another object and takes action when the state changes in some way.

- **Problem**: You have a model object with a complex state, and the state may change throughout the life of your program.
  - You want to update various other parts of the program when the object's state changes.

- **Solution**: Make the complex model object observable.

- **observable object**: An object that allows observers to examine it (notifies its observers when its state changes).
  - Permits customizable, extensible event-based behavior for data modeling and graphics.
Benefits of observer

• Abstract coupling between subject and observer; each can be extended and reused individually.

• Dynamic relationship between subject and observer; can be established at run time (can "hot-swap" views, etc) gives more programming flexibility.

• Broadcast communication: Notification is broadcast automatically to all interested objects that subscribed to it.

• Can be used to implement model-view separation in Java easily.
Observer sequence diagram

- View
  - key, mouse, or action event
  - perform appropriate action on model
  - repaint to show new model state

- Controller (optional)
  - perform appropriate action

- Model
  - notifyObservers(arg)
  - update(arg)
Observer interface

```java
// import java.util.*;

public interface Observer {
    public void update(Observable o, Object arg);
}

public class Observable {
    ...
}
```

- Basic idea:
  - Make your view code implement `Observer`.
  - Make your main model class extend `Observable`.
  - Attach the view to the model as an observer.
  - The view's `update` method will be called when the observable model changes, so write code to handle the change inside `update`. 
# Observable class

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addObserver (Observer)</td>
<td>adds an Observer to this object; its update method is called when notifyObservers is called</td>
</tr>
<tr>
<td>deleteObserver (Observer)</td>
<td>removes an Observer from this object</td>
</tr>
<tr>
<td>notifyObservers()</td>
<td>inform all observers about a change to this object; can pass optional object with more information</td>
</tr>
<tr>
<td>notifyObservers (arg)</td>
<td></td>
</tr>
<tr>
<td>setChanged()</td>
<td>flags that this object's state has changed; <em>must</em> be called prior to each call to notifyObservers</td>
</tr>
</tbody>
</table>
Multiple views

- Make an Observable model.

- Write an abstract View superclass which is a JComponent.
  - make View an observer

- Extend View for all of your actual views.
  - Give each its own unique inner components and code to draw the model's state in its own way.

- Provide a mechanism in GUI to set the view (perhaps via menus).
  - To set the view, attach it to observe the model.
Multiple views examples

- File explorer (icon view, list view, details view)
- Games (overhead view, rear view, 3D view)
- Graphs and charts (pie chart, bar chart, line chart)
Model/view exercise

- Let's develop a graphical game of Rock-Paper-Scissors.
  - Write a GUI for the game using Swing.
  - Represent the game state as a model separate from the view.
  - Make the model observable and make the view observe it.