Programming Languages and Techniques (CIS120)

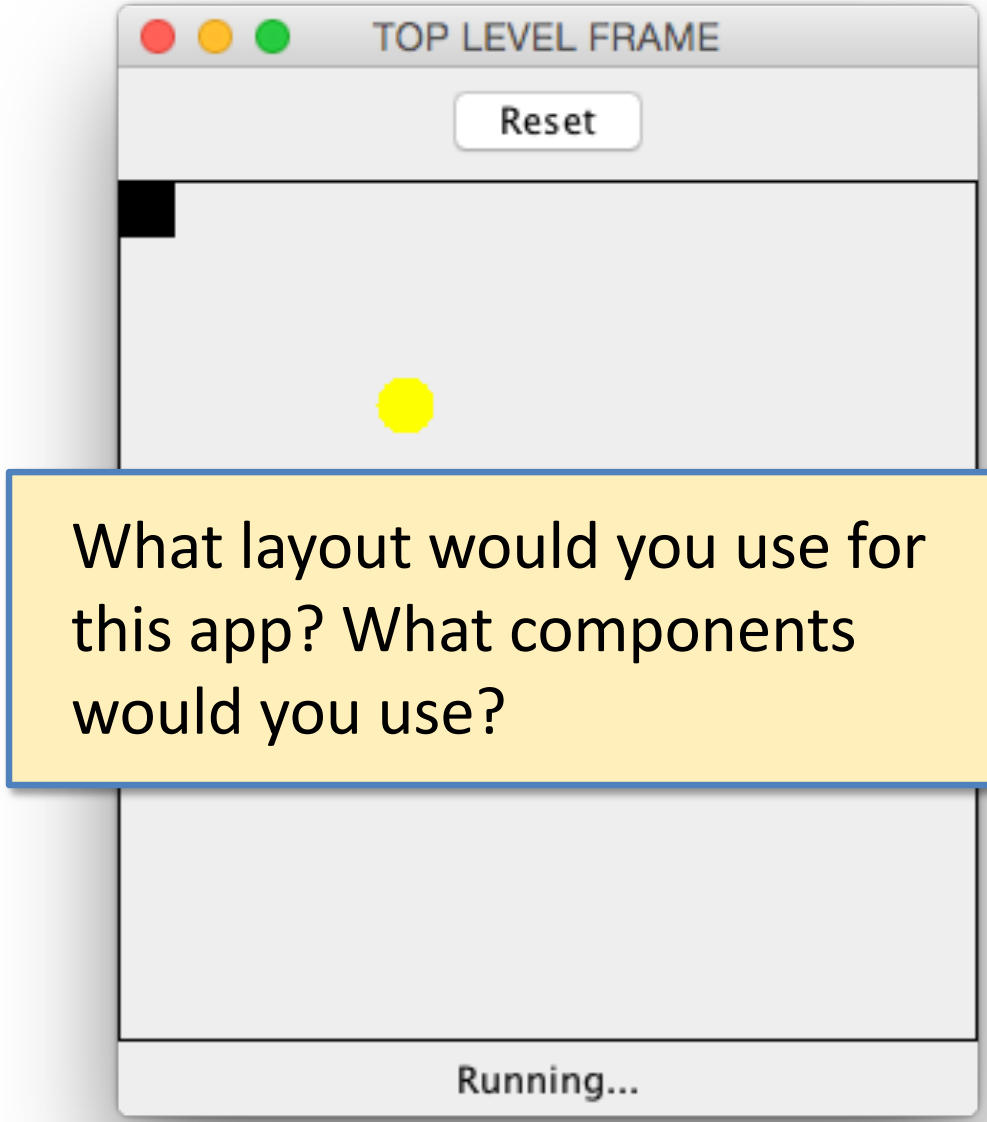
Lecture 36

Swing III: Adapters, Layout, and Mushroom of Doom

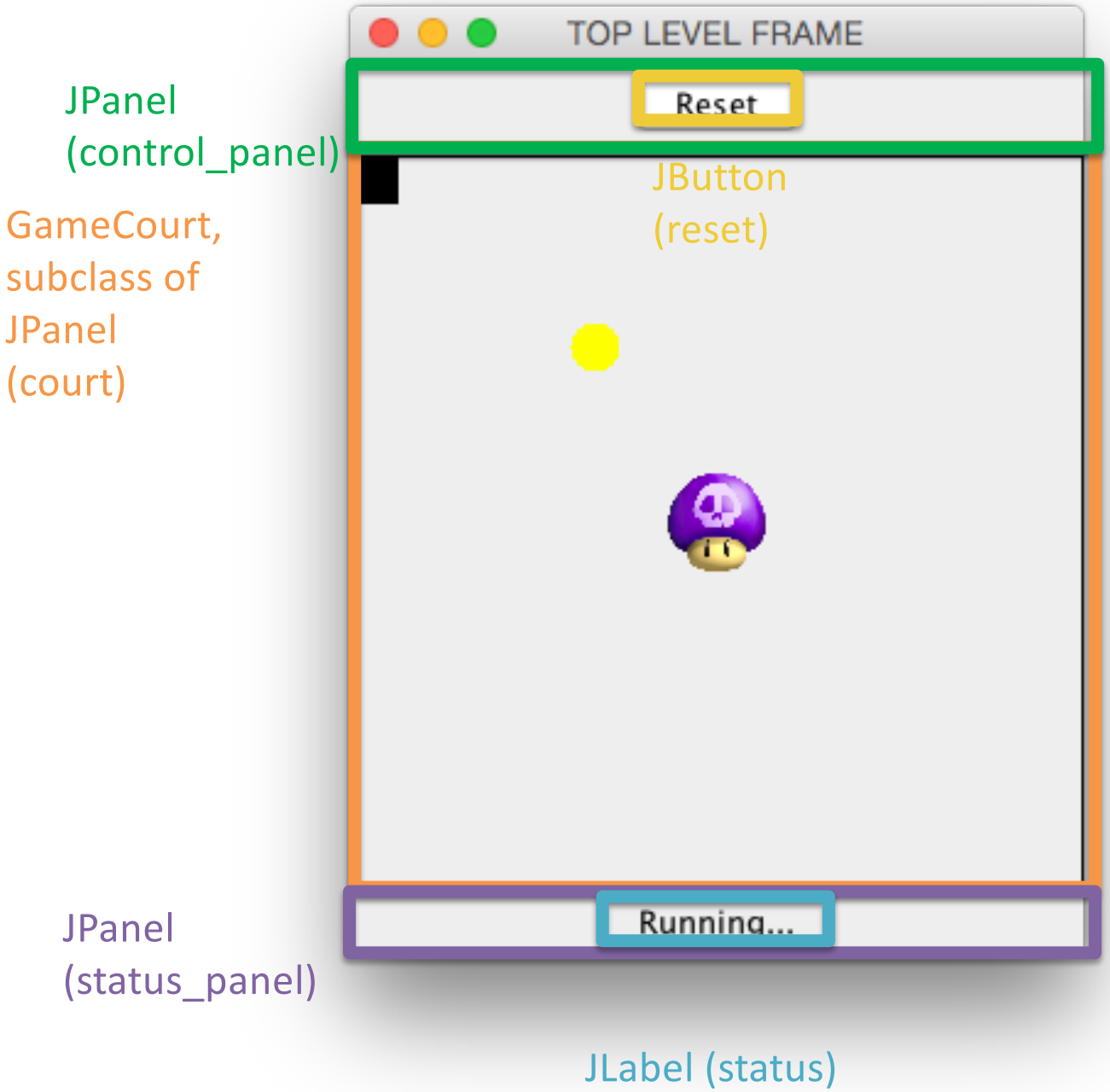
Chapter 30

Mushroom of Doom

How do we put Swing components together to make a complete game?



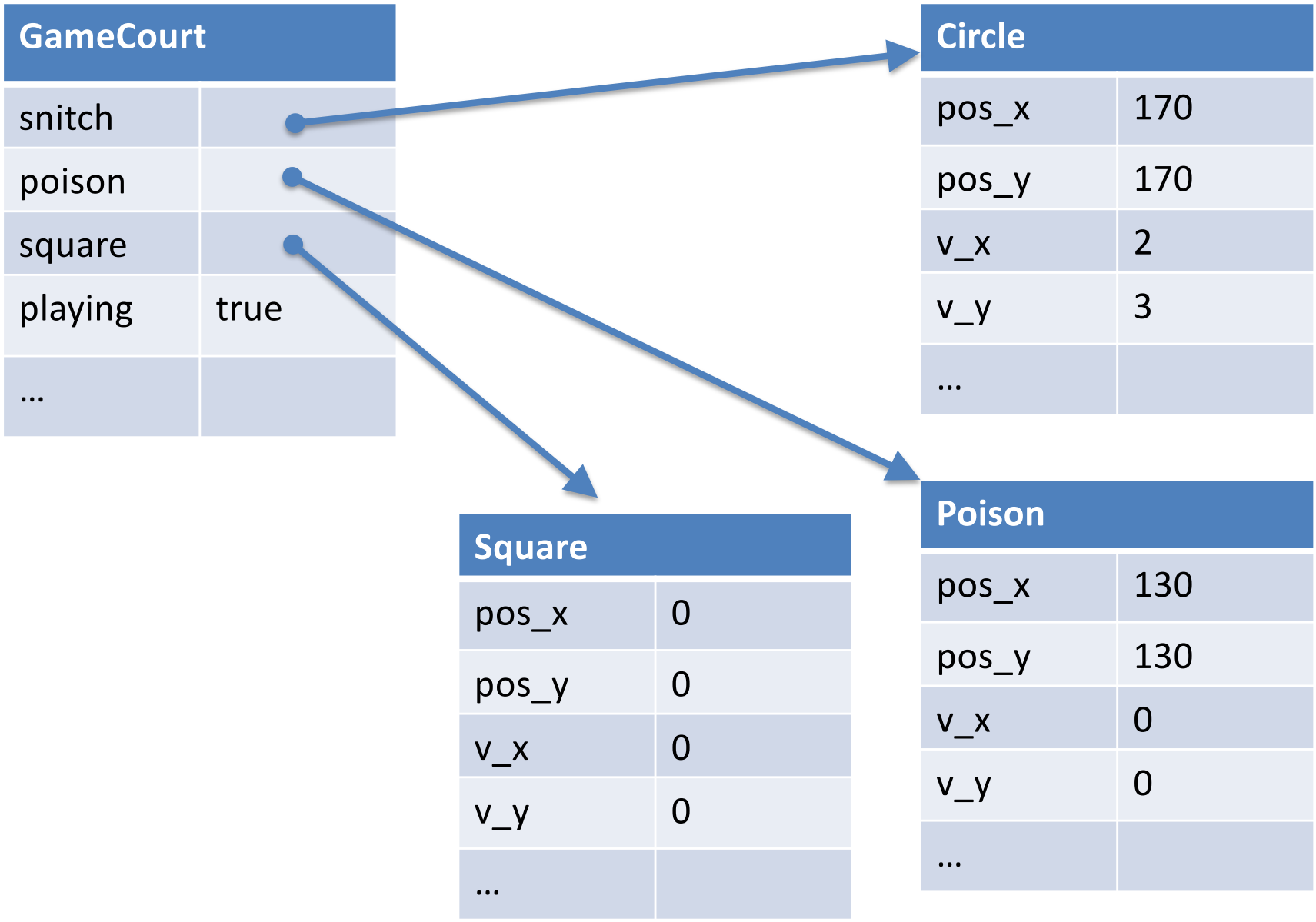
What layout would you use for this app? What components would you use?

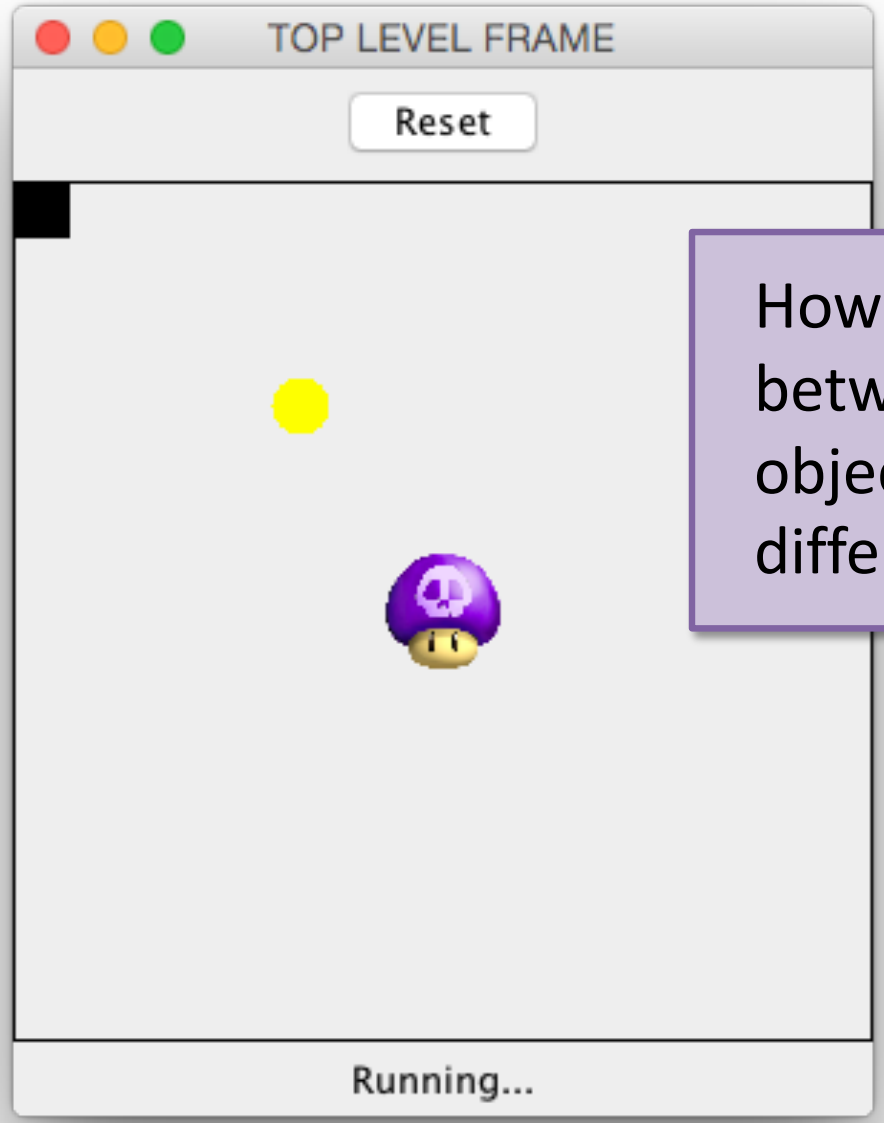

JPanel
(control_panel)
GameCourt,
subclass of
JPanel
(court)
JPanel
(status_panel)
TOP LEVEL FRAME
Reset
JButton
(reset)
Running...
JLabel (status)

Mushroom of Doom

What state should the game store?

Game State


GameCourt
snitch
poison
square
playing
true
…
Circle
pos_x
170
pos_y
170
v_x
2
v_y
3
…
Square
pos_x
0
pos_y
0
v_x
0
v_y
0
…
Poison
pos_x
130
pos_y
130
v_x
0
v_y
0
…



How can we share code between the game objects, but show them differently?

Abstract Classes

- An abstract class provides an *incomplete* implementation:
  - some methods are marked as abstract
  - those methods must be overridden to create instances

```
public abstract class AbstractClass {
    private int x = 0;
    public int m() {
        return frob(frob(x));
    }
    abstract int frob(int x);
}

class ConcreteClass extends AbstractClass {
    @Override
    int frob(int x) {
        return x * 120;
    }
}
```

Keyword "abstract" marks methods without implementations.

A subclass overrides the abstract method with an implementation.

True or False: It is possible to fill in the hole marked __??__ so that, when run, the variable ac will contain a new object of type AbstractClass.

```
public abstract class AbstractClass {
    private int x = 0;
    public int m() {
        return frob(frob(x));
    }
    abstract int frob(int x);
}

// somewhere in main:
AbstractClass ac = new AbstractClass __??__;
```

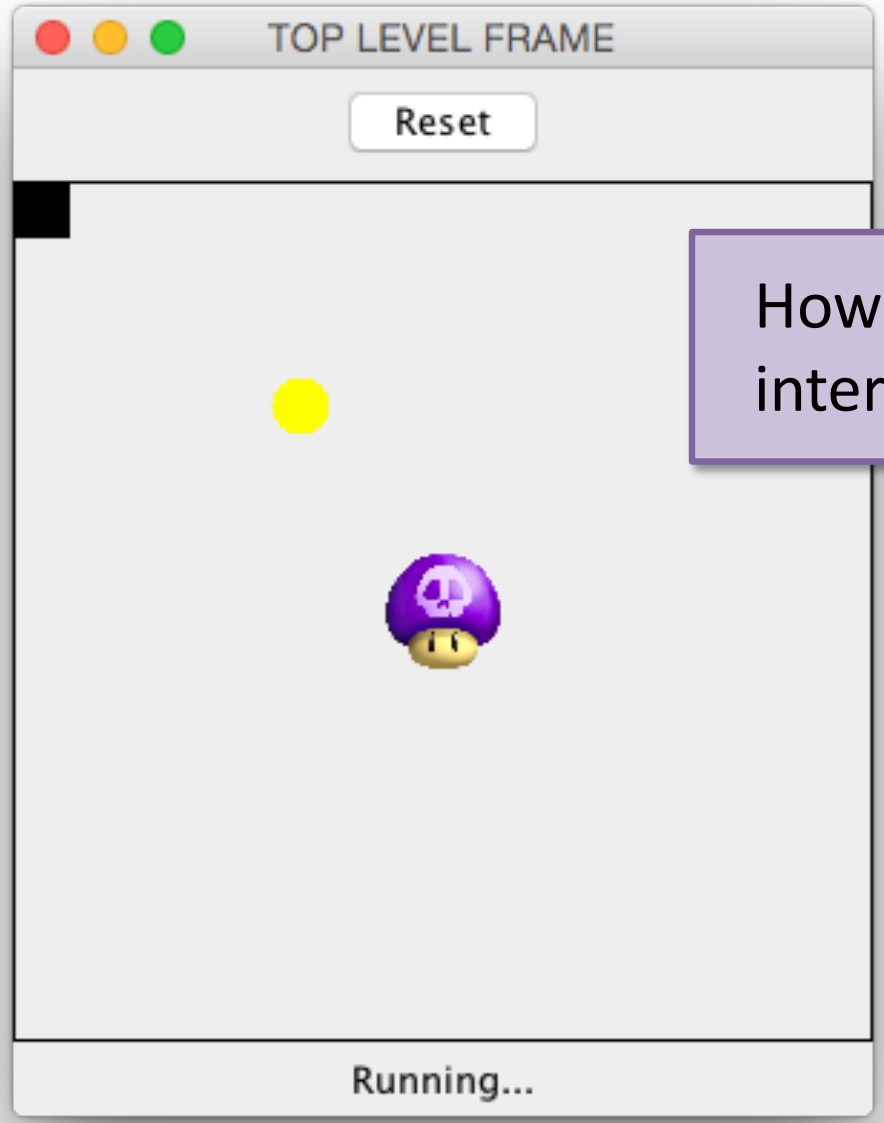
True or False: It is possible to fill in the hole marked __??__ so that, when run, the variable ac will contain a new object of type AbstractClass.

```
public abstract class AbstractClass {
    private int x = 0;
    public int m() {
        return frob(frob(x));
    }
    abstract int frob(int x);
}

// somewhere in main:
Abstract Class ac = new AbstractClass () {
    @Override
    int frob(int x) { return 0; }
};
```

Answer: True – use an anonymous inner class!

Updating the Game State: timer

```
void tick() {
  if (playing) {
    square.move();
    snitch.move();
    snitch.bounce(snitch.hitWall());      // bounce off walls...
    snitch.bounce(snitch.hitObj(poison)); // ...and the mushroom

    if (square.intersects(poison)) {
       playing = false;
       status.setText("You lose!");
    } else if (square.intersects(snitch)) {
       playing = false;
       status.setText("You win!");
    }
    repaint();
  }
}
```



How does the user interact with the game?

1. Clicking Reset button restarts the game
2. Holding arrow key makes square move
3. Releasing key makes square stop

Updating the Game State: keyboard

```
setFocusable(true);
addKeyListener(new KeyAdapter() {
  public void keyPressed(KeyEvent e) {
    if (e.getKeyCode() == KeyEvent.VK_LEFT)
        square.v_x = -SQUARE_VELOCITY;
    else if (e.getKeyCode() == KeyEvent.VK_RIGHT)
        square.v_x = SQUARE_VELOCITY;
    else if (e.getKeyCode() == KeyEvent.VK_DOWN)
        square.v_y = SQUARE_VELOCITY;
    else if (e.getKeyCode() == KeyEvent.VK_UP)
        square.v_y = -SQUARE_VELOCITY;
    }

    public void keyReleased(KeyEvent e) {
        square.v_x = 0;
        square.v_y = 0;
    }
});
```

Allow the court to handle key events

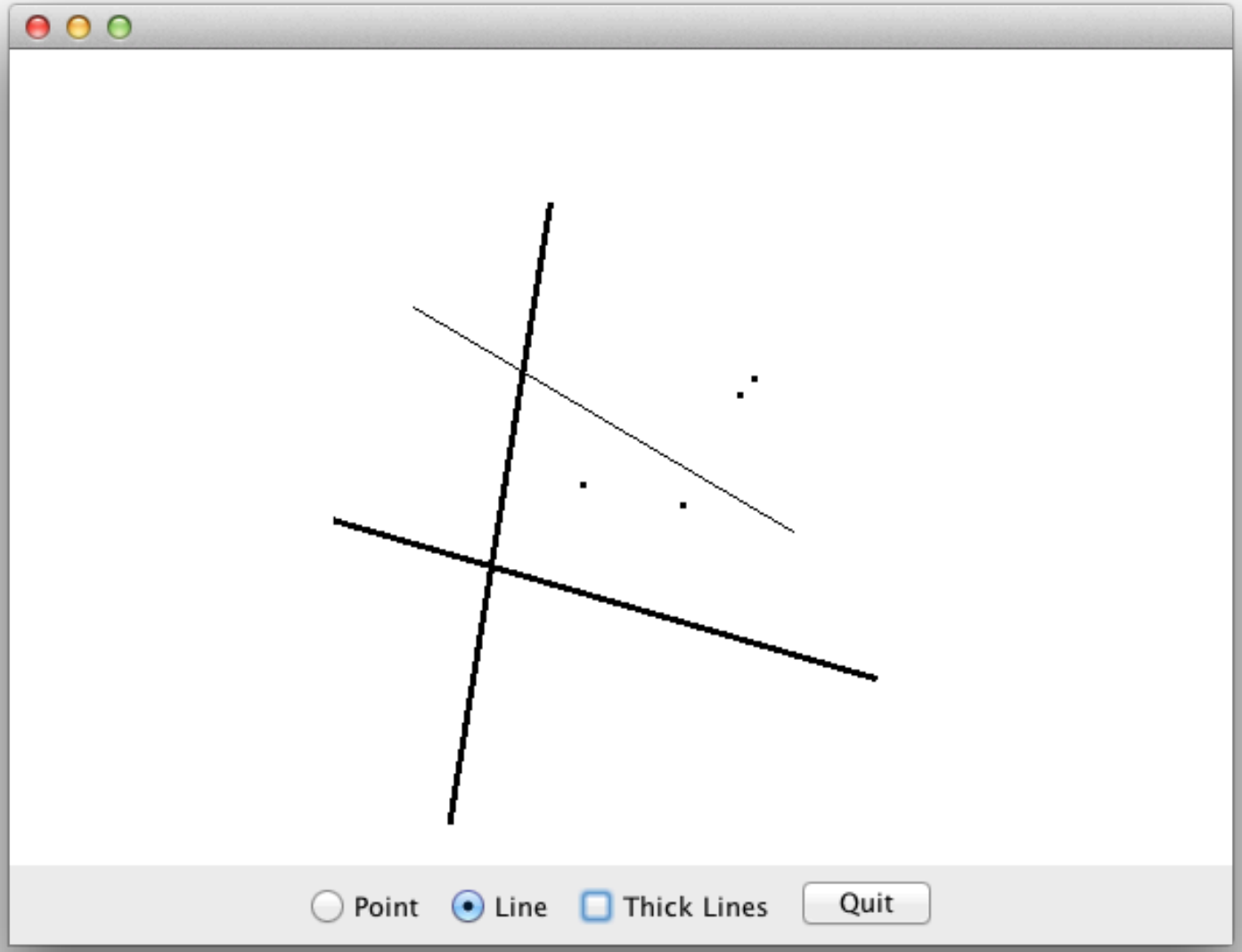
Make square's velocity nonzero when a key is pressed

Make square's velocity zero when a key is released

Paint Revisited

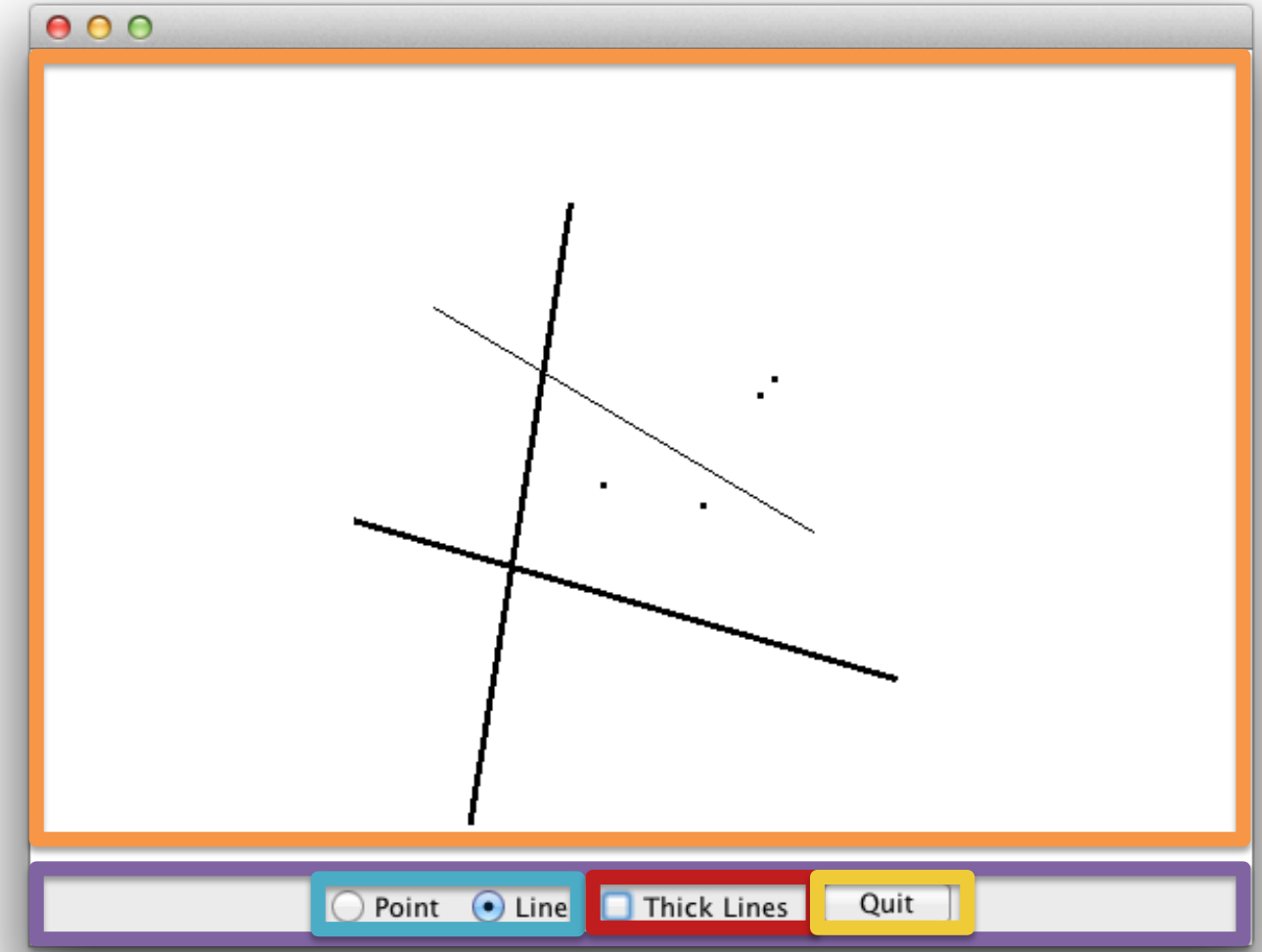
Using Anonymous Inner Classes

Refactoring for OO Design



What layout would you use for this app? What components would you use?

Canvas
subclass of
JPanel
(canvas)



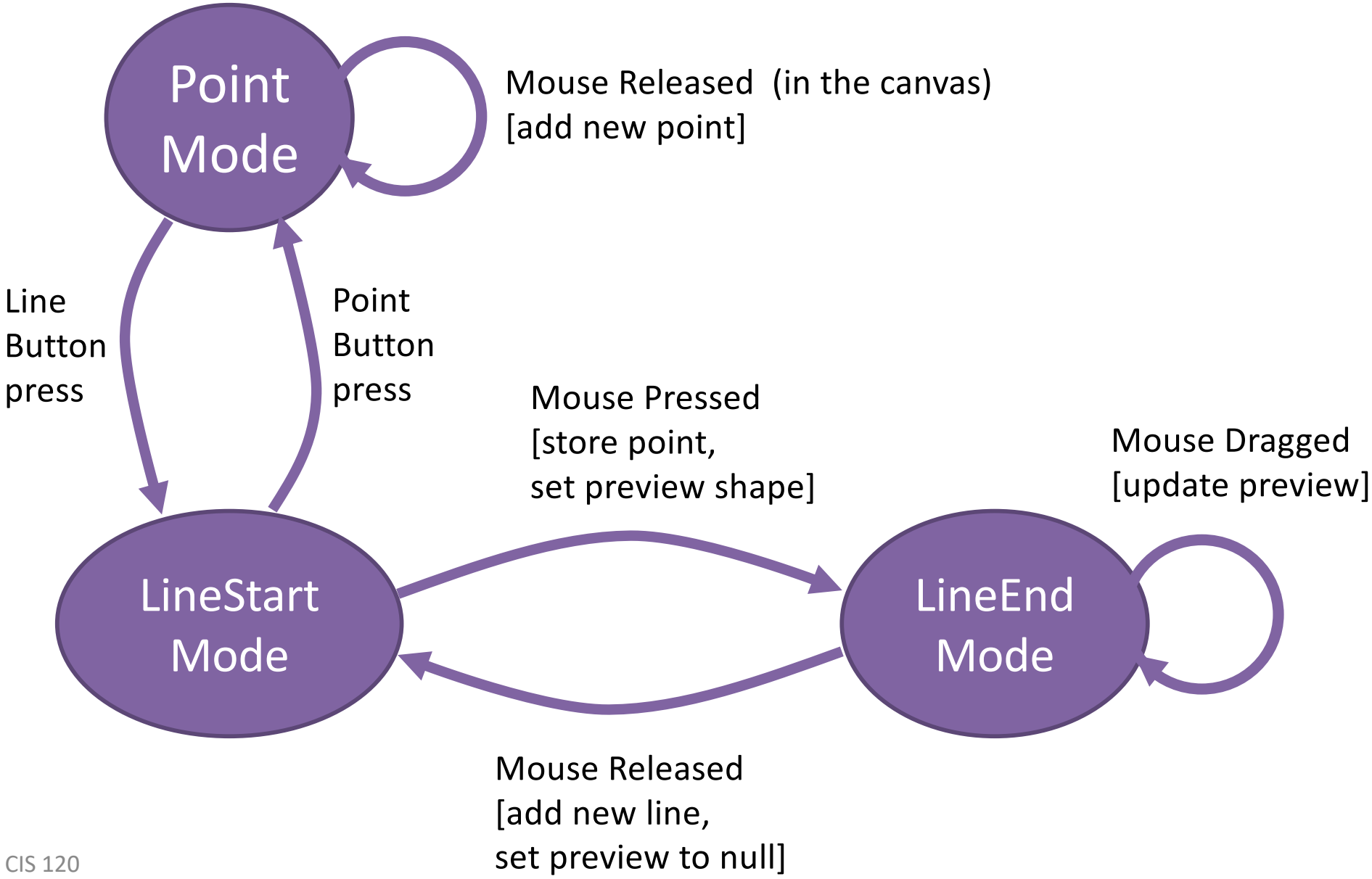
JPanel
(toolbar)

JRadioButton
(point, line)

JCheckbox
(thick)

JButton
(quit)

Mouse Interaction in Paint



Paint Revisited
(thoroughly discussed in Chap 31)

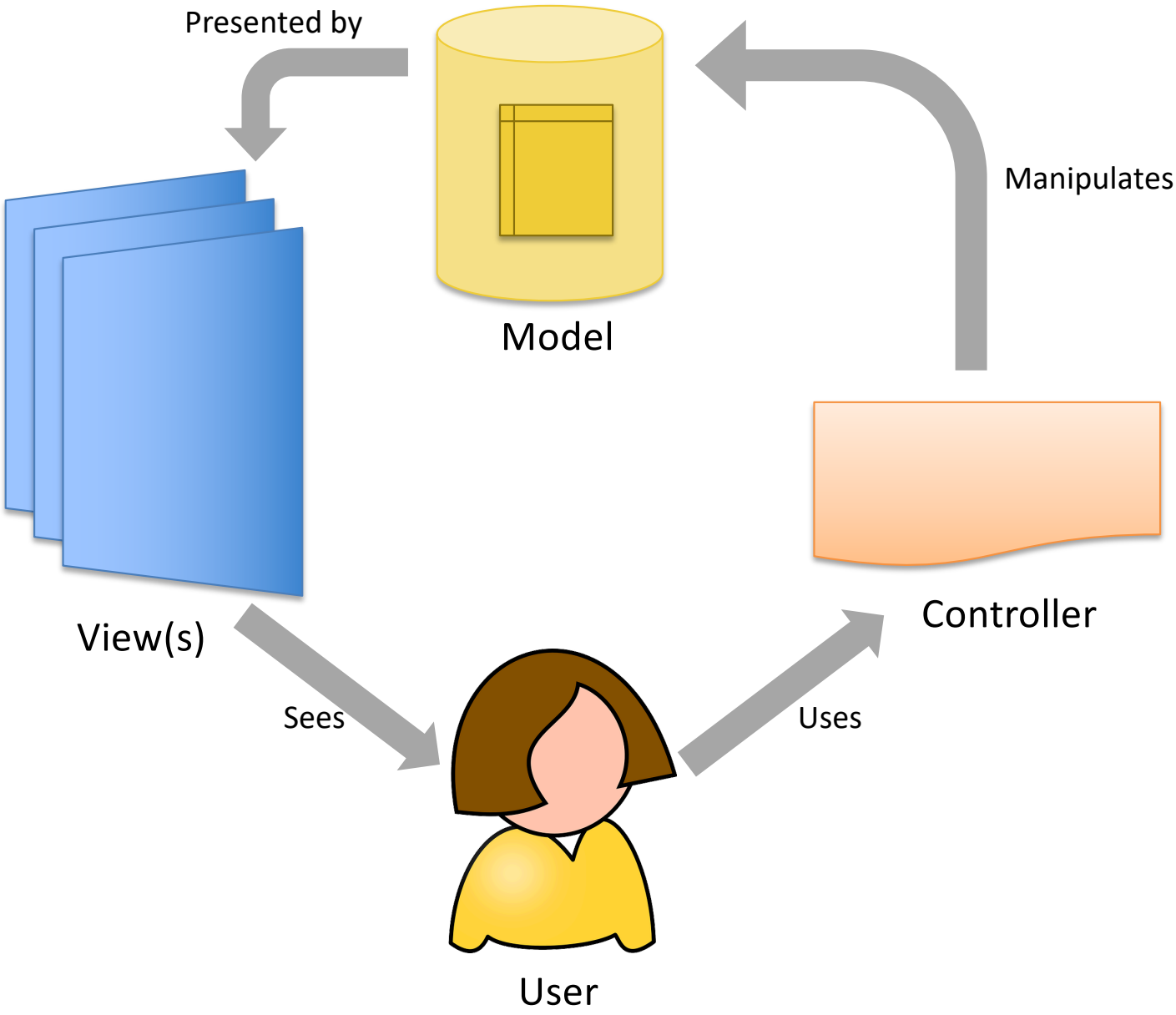
Using Anonymous Inner Classes

Refactoring for OO Design

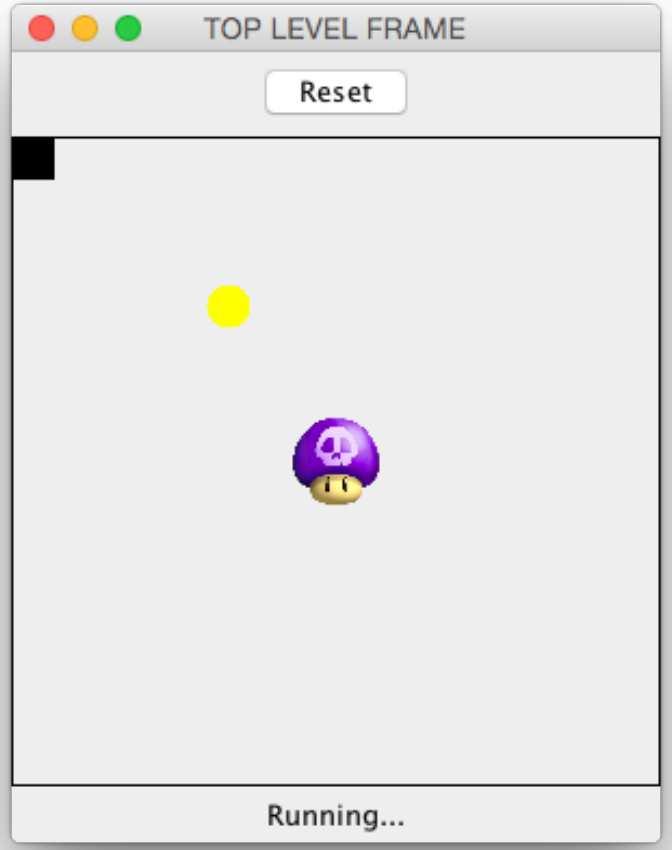
(See PaintA.java … PaintE.java)

Model View Controller Design Pattern

MVC Pattern


Presented by
Model
Manipulates
View(s)
Controller
Sees
Uses
User

Example 1: Mushroom of Doom



Example: MOD Program Structure

- GameCourt, GameObj + subclass local state
  - object location & velocity
  - status of the game (playing, win, loss)
  - how the objects interact with eachother (tick)

Model

- Draw methods
  - paintComponent in GameCourt
  - draw methods in GameObj subclasses
  - status label

View

- Game / GameCourt
  - Reset button (updates model)
  - Keyboard control (updates square velocity)

Controller

Example: Paint Program Structure

- Main frame for application (class Paint)
  - List of shapes to draw
  - The current color
  - The current line thickness

Model

- Drawing panel (class Canvas, inner class of Paint)

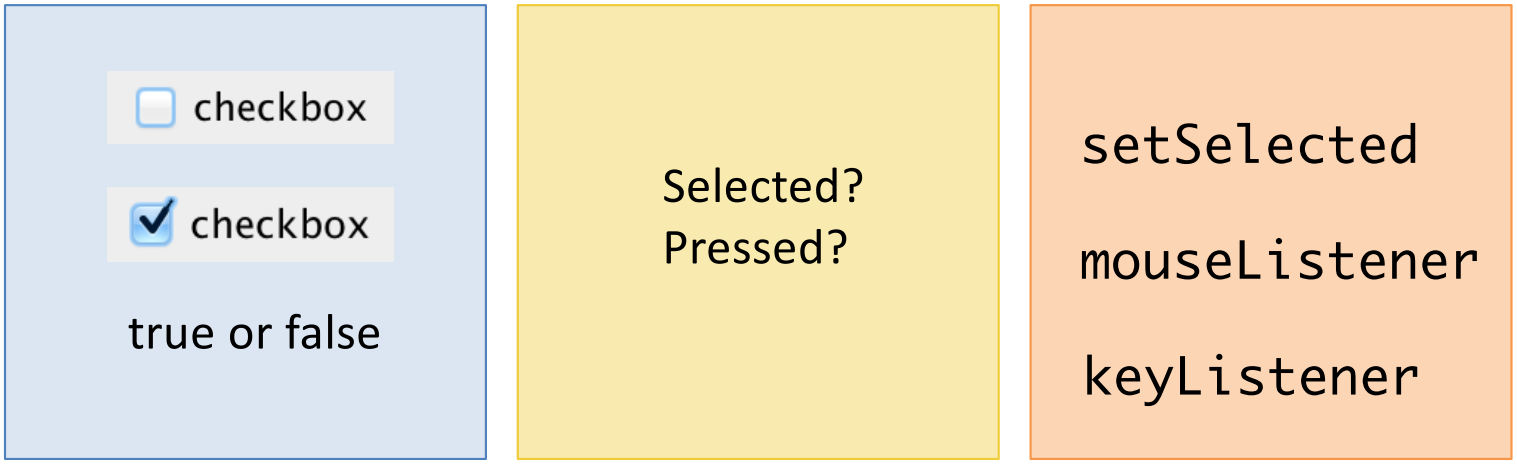
View

- Control panel (class JPanel)
  - Contains radio buttons for selecting shape to draw
  - Line thickness checkbox, undo and quit buttons

Controller

- Connections between Preview shape (if any…)
  - Preview Shape: View <-> Controller
  - MouseAdapter: Controller <-> Model

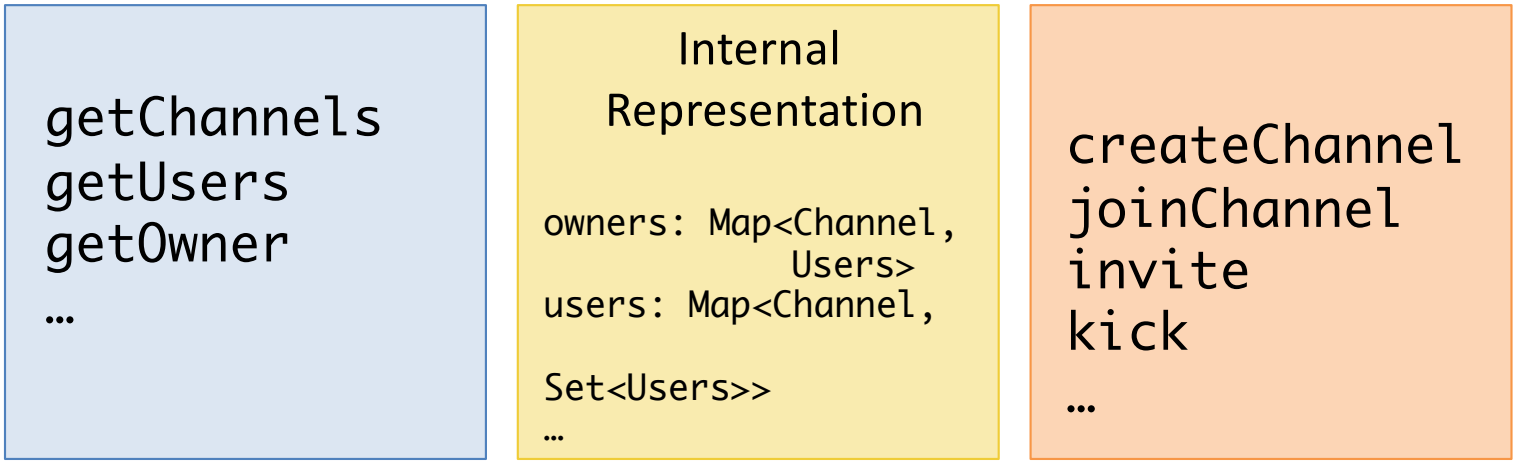
Example: CheckBox




```
Class JToggleButton.ToggleButtonModel

boolean   isSelected()
void   setPressed(boolean b)
void   setSelected(boolean b)
```

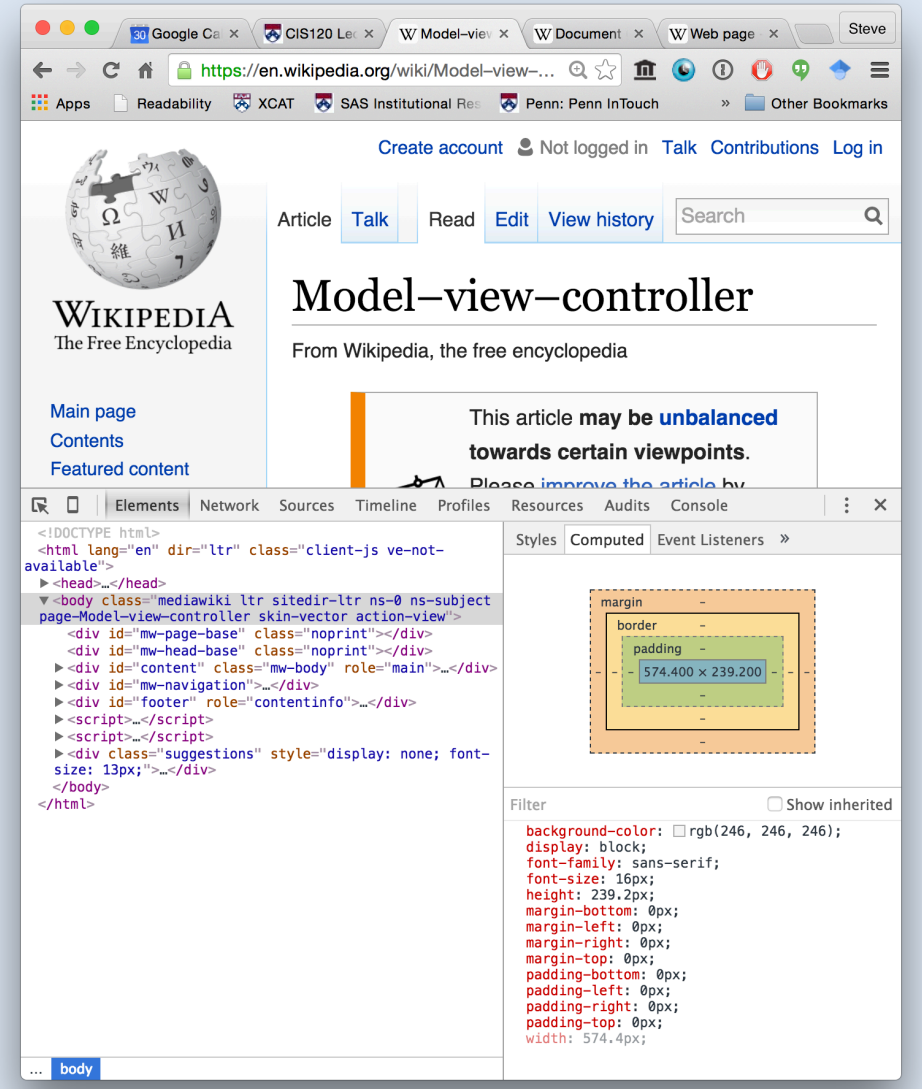
Checks if the button is selected.
Sets the pressed state of the button.
Sets the selected state of the button.

Example: Chat Server



ServerModel

Example: Web Pages



Views

Internal Representation: DOM (Document Object Model)

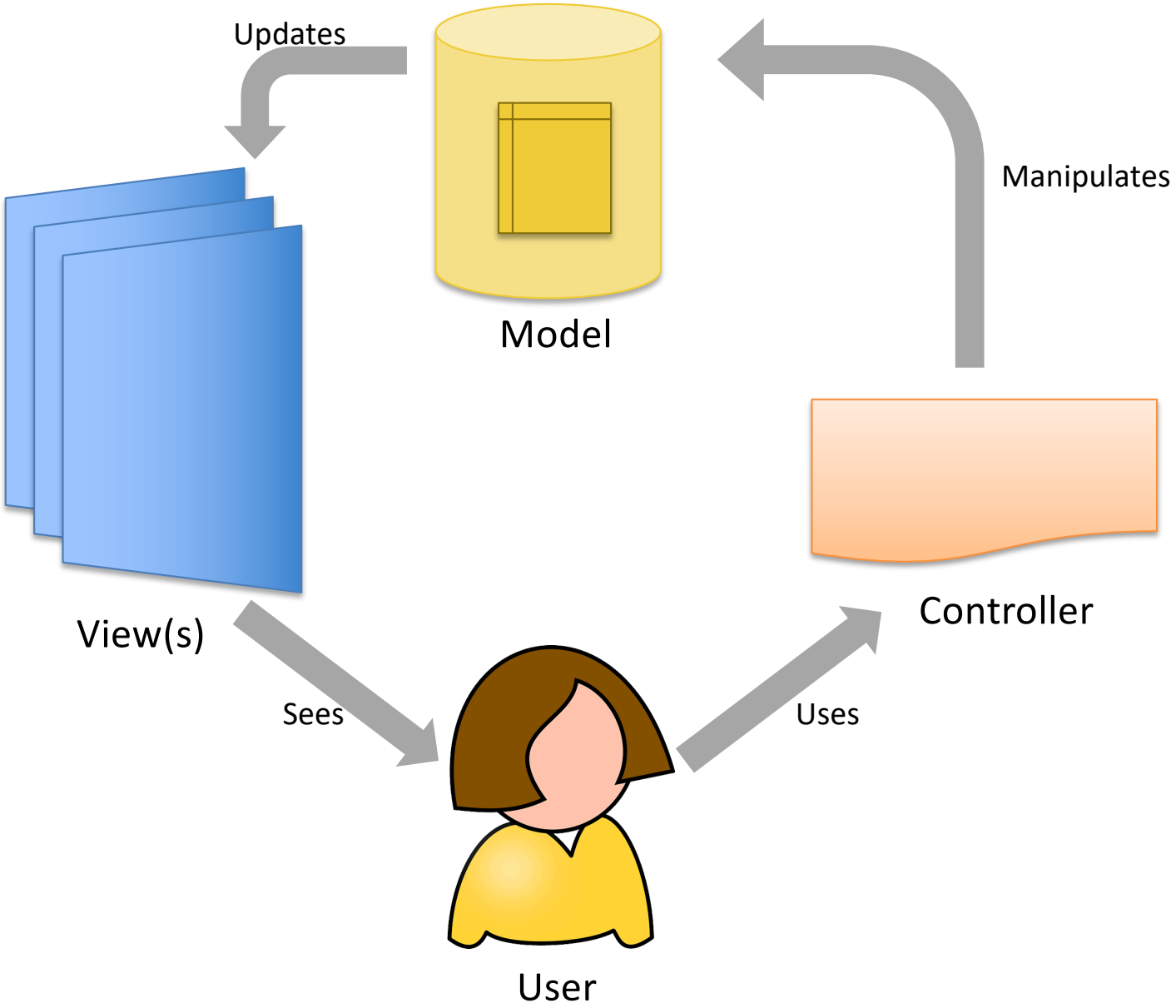
Model

JavaScript API

```
document.
addEventListener()
```

Controllers

MVC Pattern


Updates
Model
Manipulates
View(s)
Controller
Sees
Uses
User

MVC Benefits?

- Decouples important "model state" from how that state is presented and manipulated
  - Suggests where to insert interfaces in the design
  - Makes the model testable independent of the GUI

- Multiple views
  - e.g. from two different angles, or for multiple different users

- Multiple controllers
  - e.g. mouse vs. keyboard interaction

MVC Variations

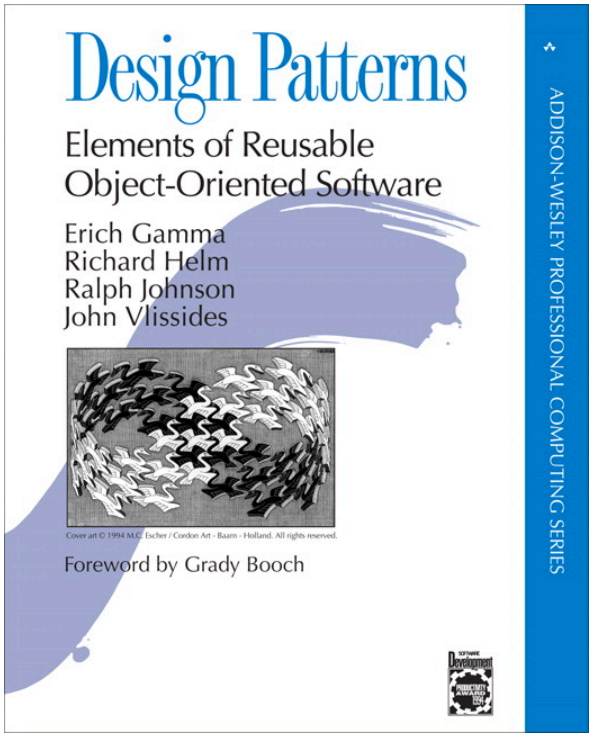
- Many variations on MVC pattern

- Hierarchical / Nested
  - As in the Swing libraries, in which JComponents often have a "model" and a "controller" part

- Coupling between Model / View or View / Controller
  - e.g. in MOD the Model and the View are coupled because the model carries most of the information about the view

Design Patterns

- Design Patterns
  - Influential OO design book published in 1994 (so a bit dated)
  - Identifies many common situations and "patterns" for implementing them in OO languages



- Some we have seen explicitly:
  - e.g. *Iterator* pattern
- Some we've used but not explicitly described:
  - e.g. The Broadcast class from the Chat HW uses the *Factory* pattern
- Some are workarounds for OO's lack of some features:
  - e.g. The *Visitor* pattern is like OCaml's fold + pattern matching