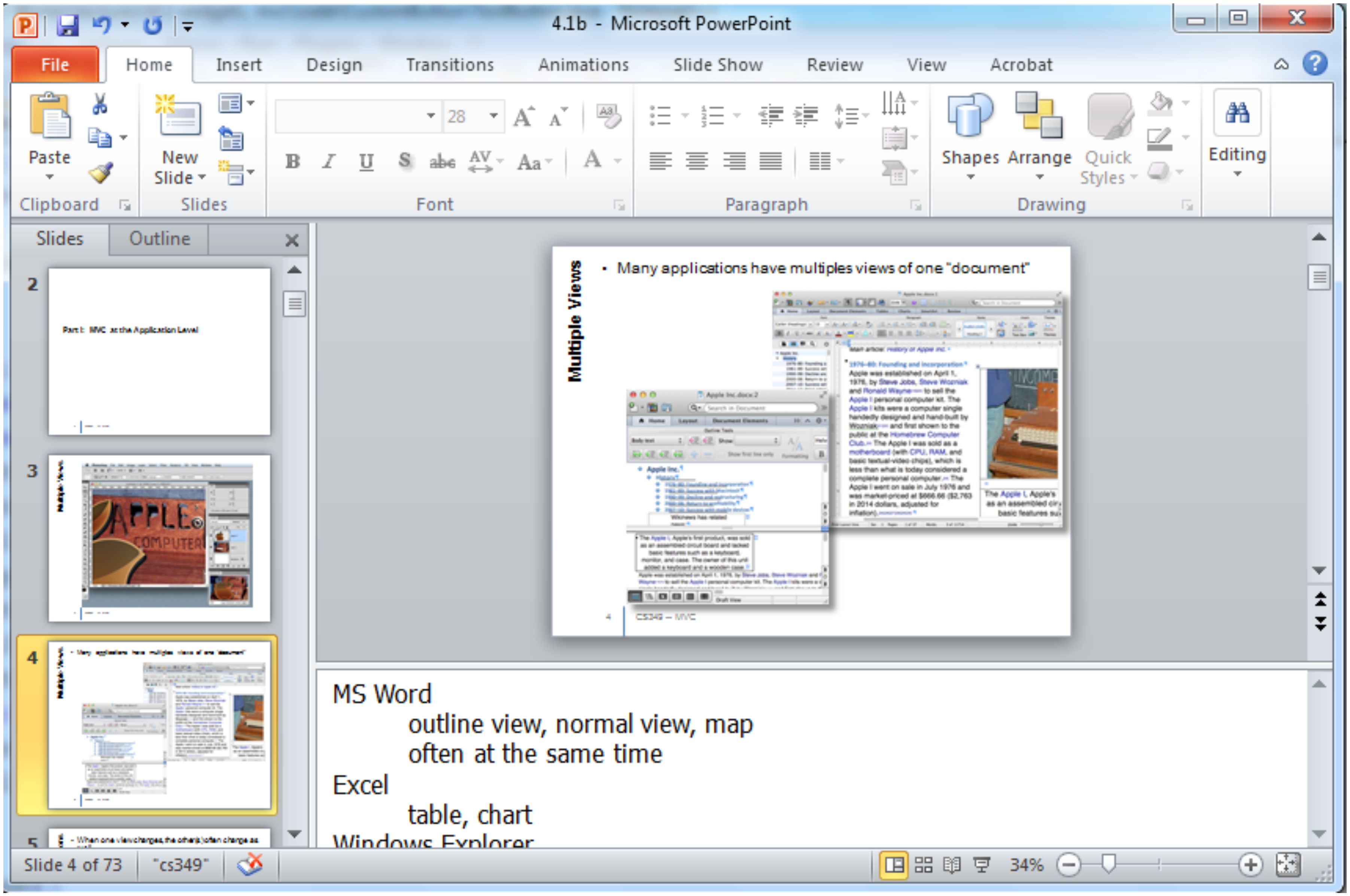
Model-View-Controller

Motivation

The MVC pattern

Using the Observer pattern in Java

MVC Rationale



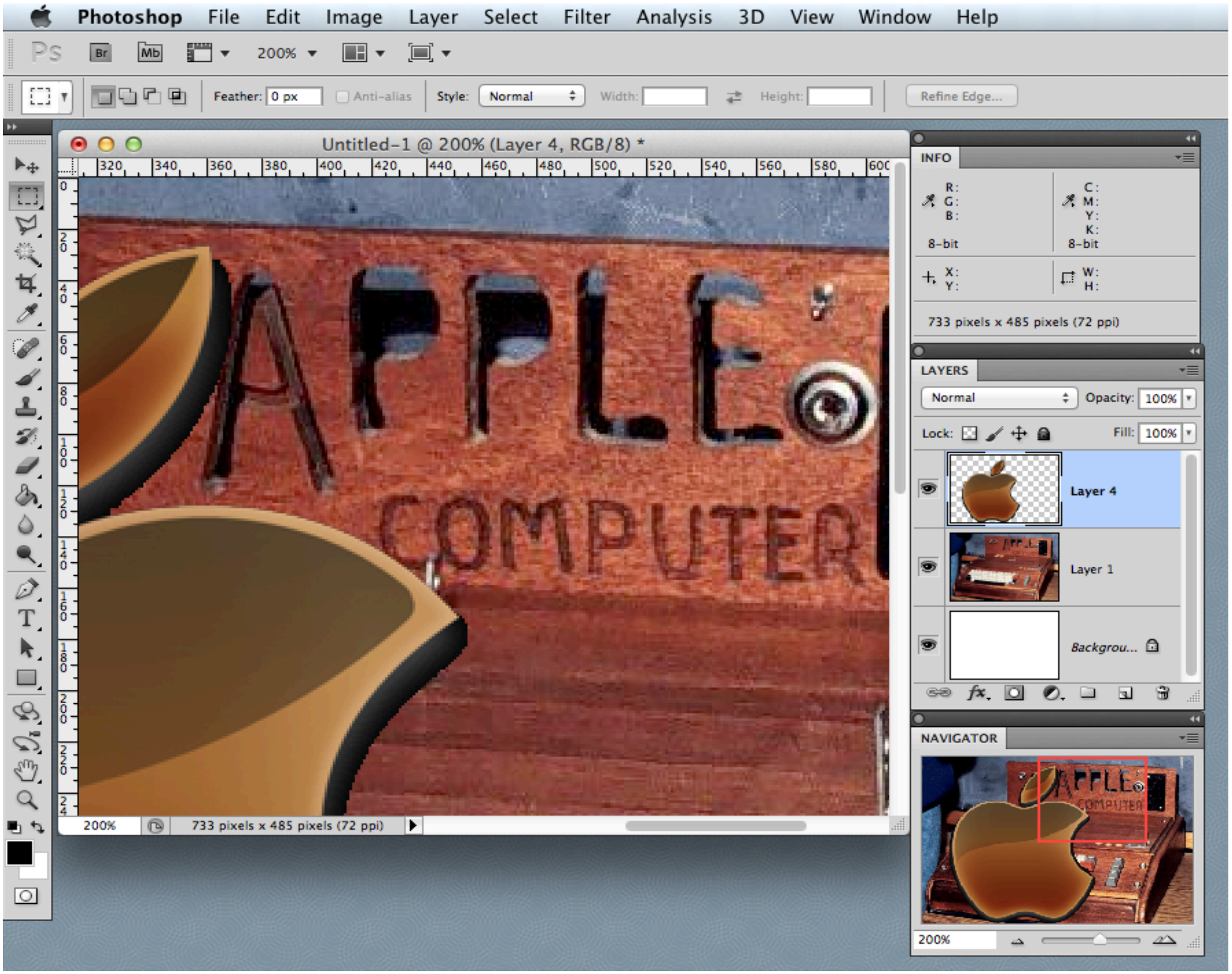
Multiple Views

• Many applications have multiples views of one "document"

MS Word
outline view, normal view, map
often at the same time
Excel
table, chart
Windows Explorer

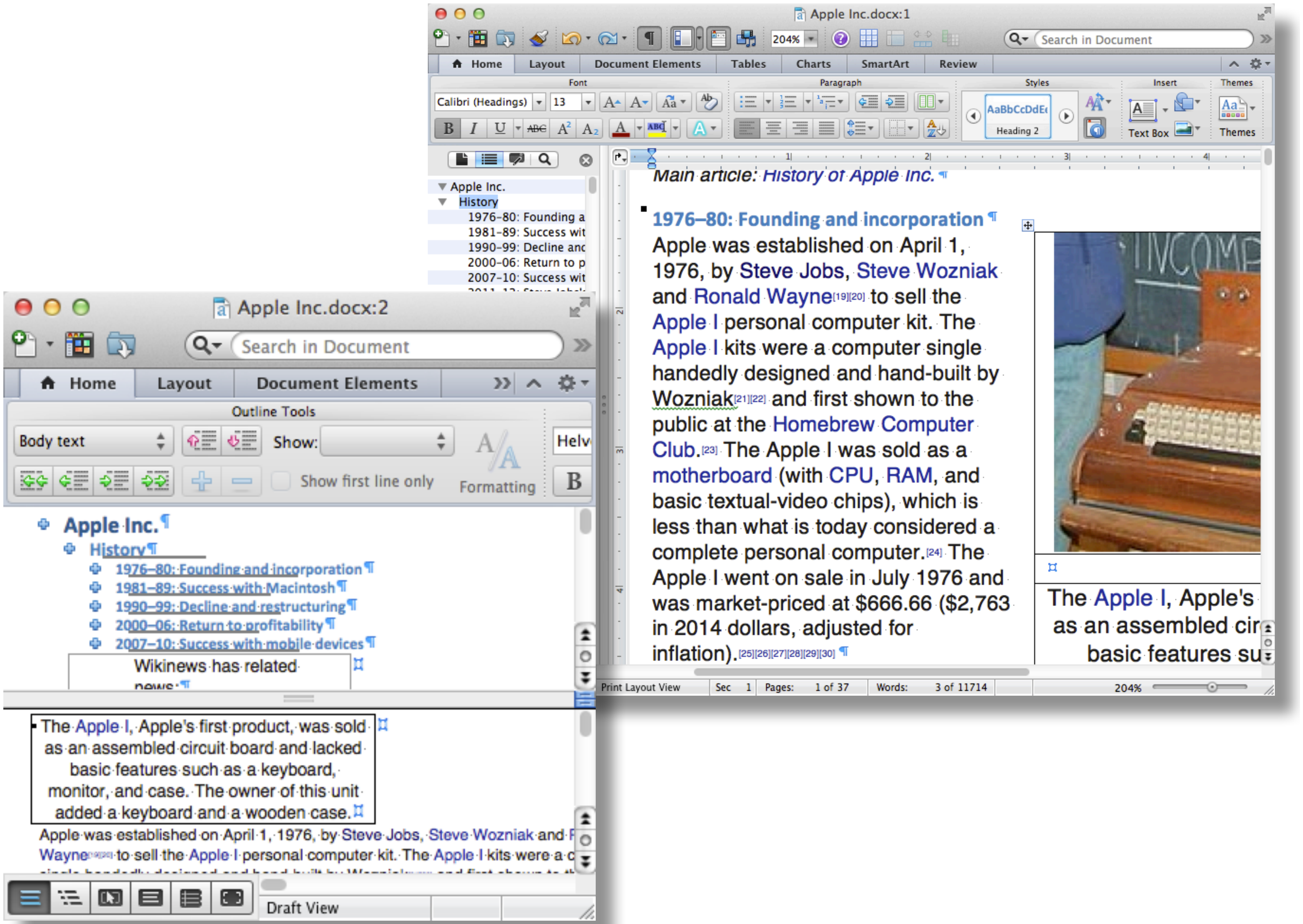
Multiple views, loosely coupled to the underlying data model.

Multiple Views



Multiple Views

Many applications have multiple views of one “document”



Main article: History of Apple Inc.

1976–80: Founding and incorporation

Apple was established on April 1, 1976, by Steve Jobs, Steve Wozniak and Ronald Wayne[19][20] to sell the Apple I personal computer kit. The Apple I kits were a computer single handedly designed and hand-built by Wozniak[21][22] and first shown to the public at the Homebrew Computer Club.[23] The Apple I was sold as a motherboard (with CPU, RAM, and basic textual-video chips), which is less than what is today considered a complete personal computer.[24] The Apple I went on sale in July 1976 and was market-priced at $666.66 ($2,763 in 2014 dollars, adjusted for inflation).[25][26][27][28][29][30]

The Apple I, Apple's as an assembled cir basic features su

Apple Inc.

History

1976–80: Founding and incorporation

1981–89: Success with Macintosh

1990–99: Decline and restructuring

2000–06: Return to profitability

2007–10: Success with mobile devices

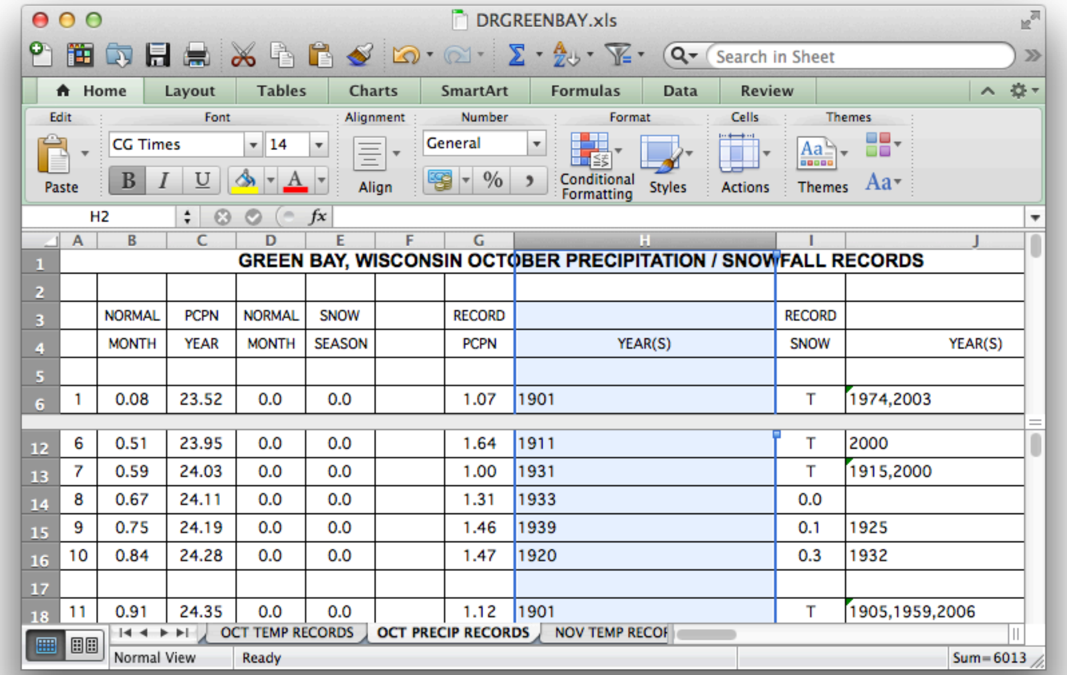
Wikinews has related news:

The Apple I, Apple's first product, was sold as an assembled circuit board and lacked basic features such as a keyboard, monitor, and case. The owner of this unit added a keyboard and a wooden case.

Apple was established on April 1, 1976, by Steve Jobs, Steve Wozniak and R Wayne to sell the Apple I personal computer kit. The Apple I kits were a c

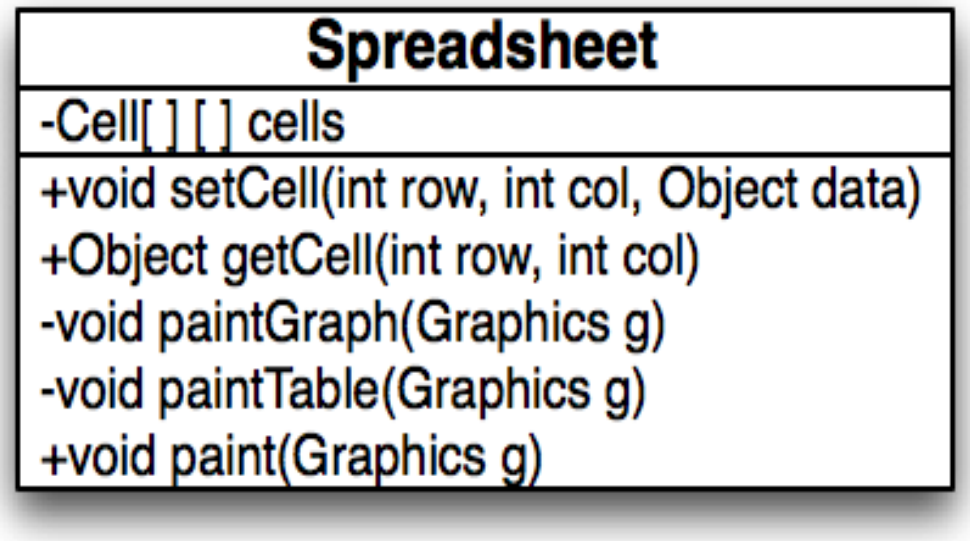
Observations

- When one view changes, other(s) often need to change.
  - Ideally, we want a single representation of the underlying data, and multiple views of that data.
- The user interface code probably changes more and faster than the underlying application
  - Many recent changes in MS Office were to UI code
  - Excel's underlying functions and data structures are probably very similar to Visicalc, the original spreadsheet
- How do we design software to support these observations?



	A	B	C	D	E	F	G	H	I	J
1	GREEN BAY, WISCONSIN OCTOBER PRECIPITATION / SNOWFALL RECORDS									
2										
3		NORMAL	PCPN	NORMAL	SNOW		RECORD		RECORD	
4		MONTH	YEAR	MONTH	SEASON		PCPN	YEAR(S)	SNOW	YEAR(S)
5										
6	1	0.08	23.52	0.0	0.0		1.07	1901	T	1974,2003
12	6	0.51	23.95	0.0	0.0		1.64	1911	T	2000
13	7	0.59	24.03	0.0	0.0		1.00	1931	T	1915,2000
14	8	0.67	24.11	0.0	0.0		1.31	1933	0.0	
15	9	0.75	24.19	0.0	0.0		1.46	1939	0.1	1925
16	10	0.84	24.28	0.0	0.0		1.47	1920	0.3	1932
17										
18	11	0.91	24.35	0.0	0.0		1.12	1901	T	1905,1959,2006

Possible Design: Tight Coupling



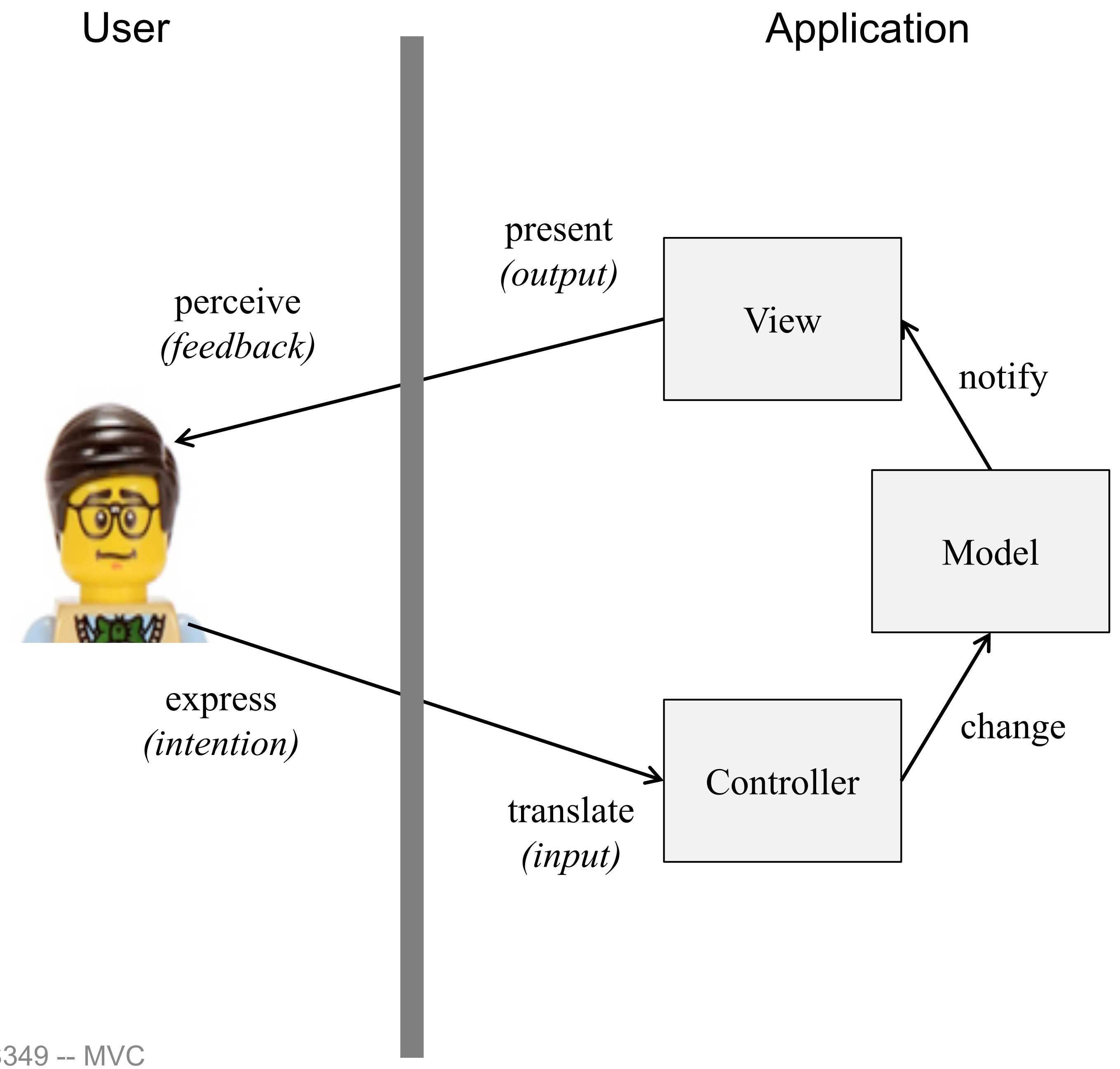
Spreadsheet

-Cell[ ] [ ] cells

+void setCell(int row, int col, Object data)
+Object getCell(int row, int col)
-void paintGraph(Graphics g)
-void paintTable(Graphics g)
+void paint(Graphics g)

- Issues with bundling everything together:
  - What if we want to display data from a different type of source (e.g. a database)?
  - What if we want to add new ways to view the data?
- Primary problem with this approach:
  - <u>Tight coupling</u> of data and presentation prevents easy modification and extension.

Solution: Model-View-Controller (MVC)



MVC History

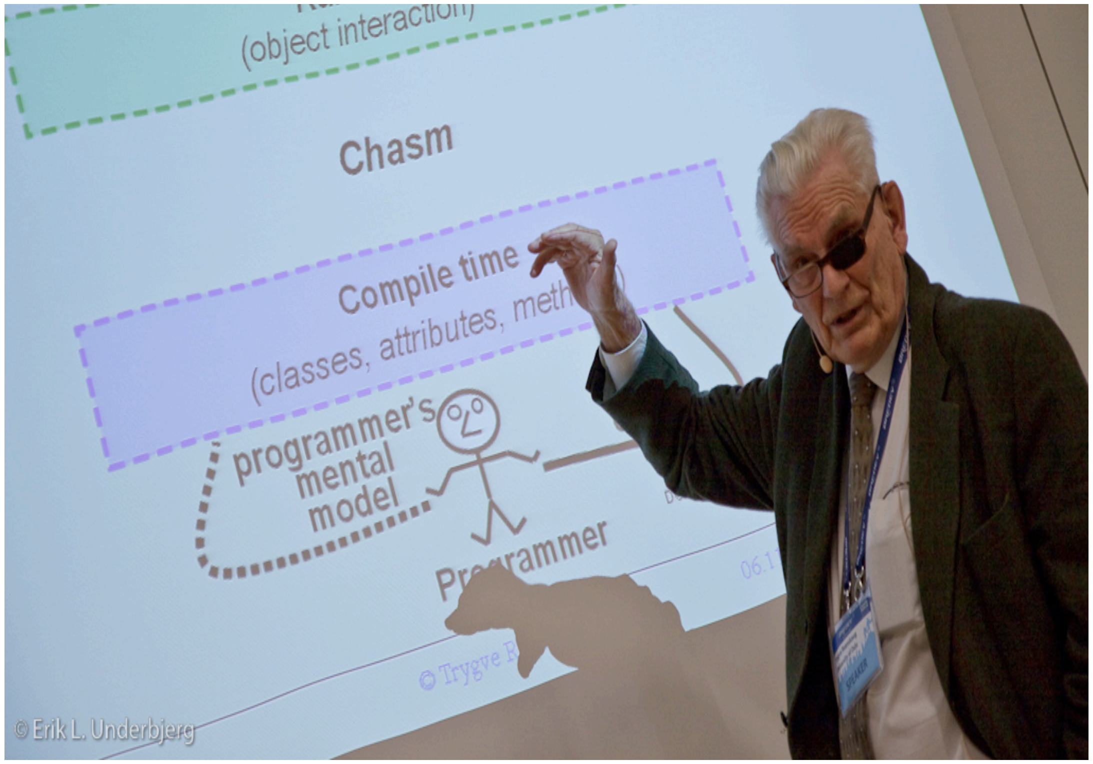
Developed for Smalltalk-80 in 1979 by Trygve Reenskaug, while visiting Xerox PARC.

Now a standard design pattern for graphical user interfaces that is used at many levels, including the overall application design and individual visual components.

Variations

- Model-View-Presenter
- Model-View-Adapter
- Hierarchical Model-View-Controller

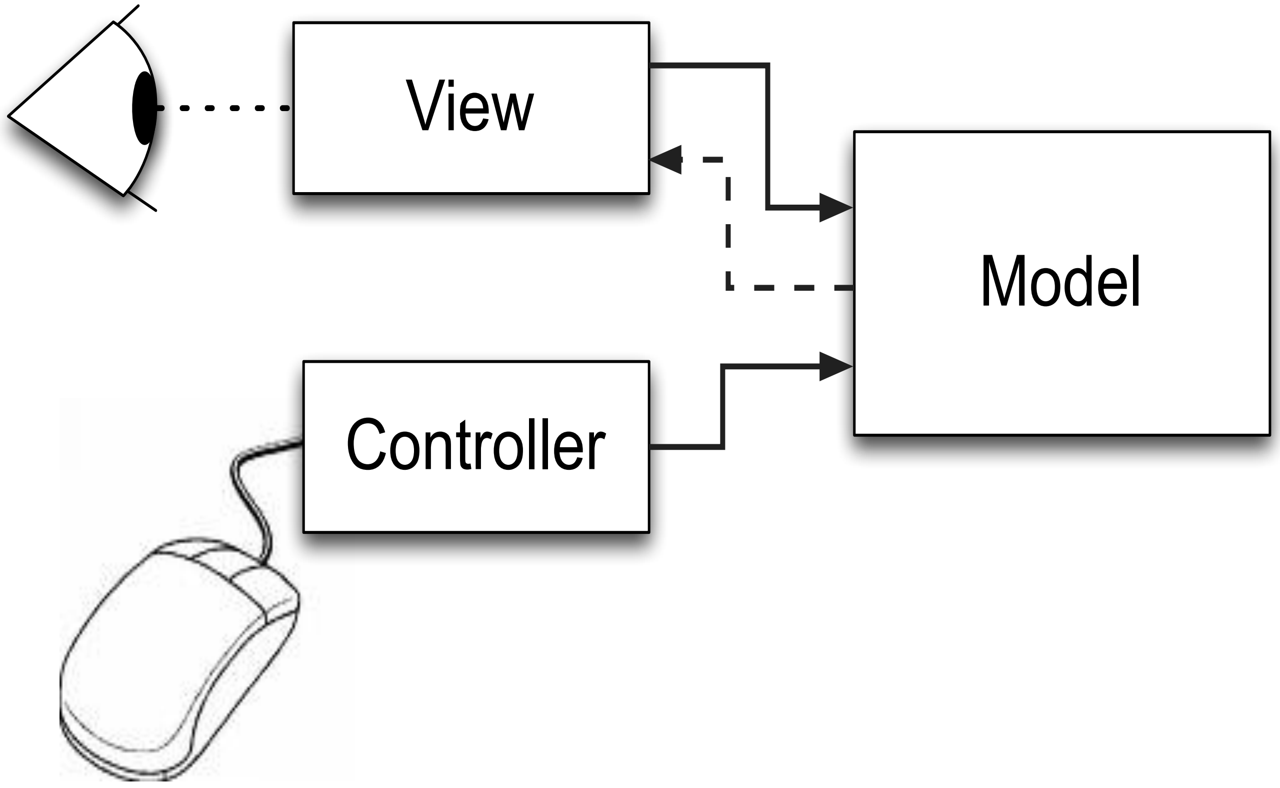
We use "standard" MVC in this course.



Solution: Model-View-Controller (MVC)

Interface architecture decomposed into three parts (classes):

- **Model**: manages the data and its manipulation
- **View**: manages the presentation of the data
- **Controller**: manages user interaction



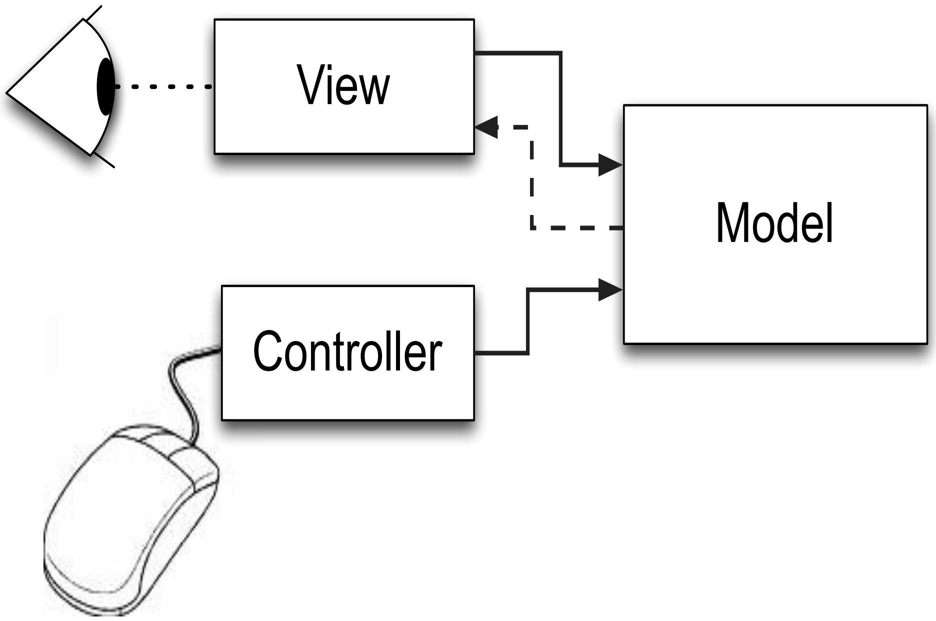
MVC Classes

These classes are loosely coupled:

- View and Controller both know about the model (through a public interface that the model defines).
  - Controller is able to update the model based on user input.
  - View needs to be able to display data from the model.
- Model only knows about the View through it's interface.
  - Notifies the view(s) when the model's internal state changes.

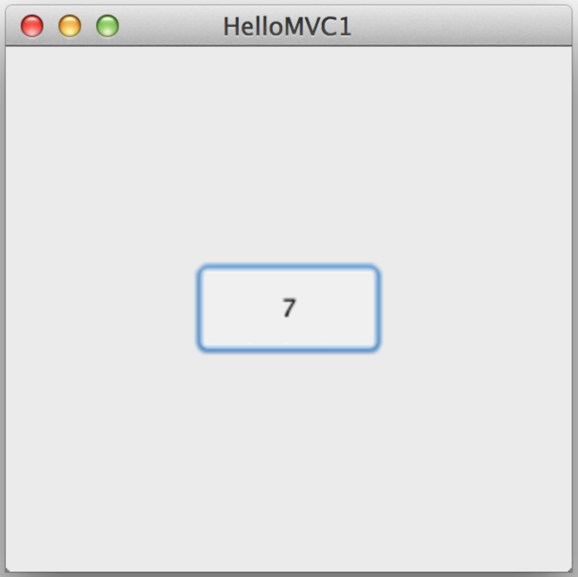
View Interface

```
interface IView {
    public void updateView();
}
```

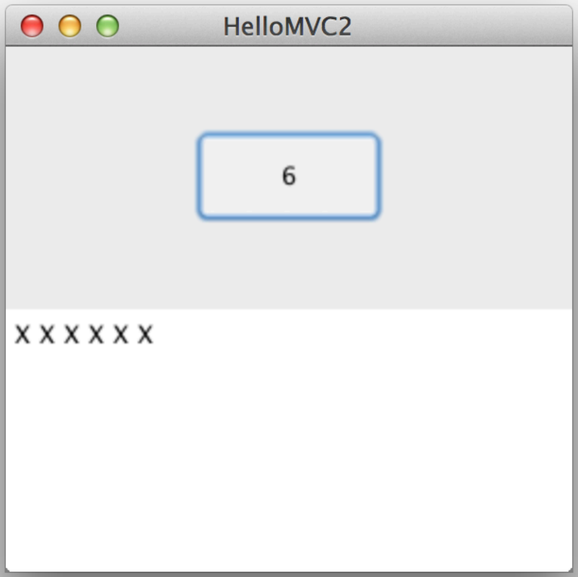


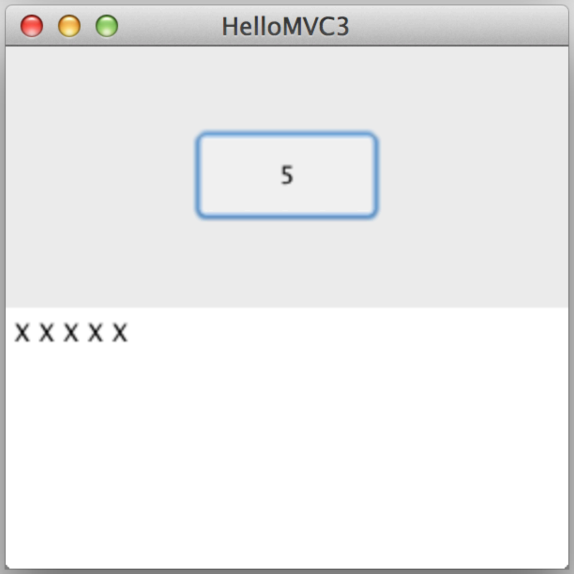
HelloMVC1 to HelloMVC4 Code Examples

HelloMVC1
1 view



HelloMVC2
2 (or more) views



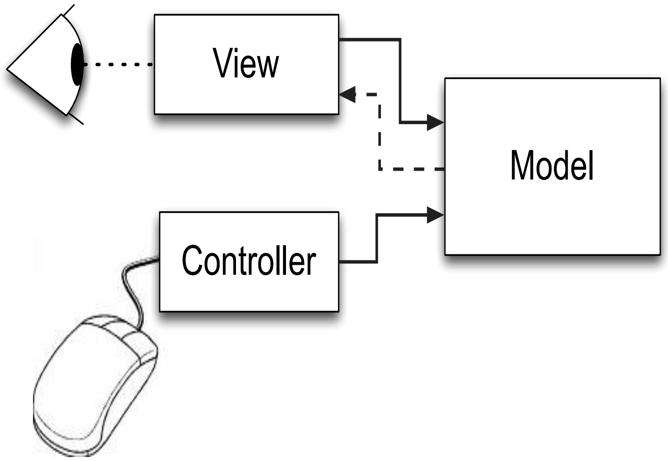


HelloMVC3
Includes anonymous inner classes, inner classes, etc.

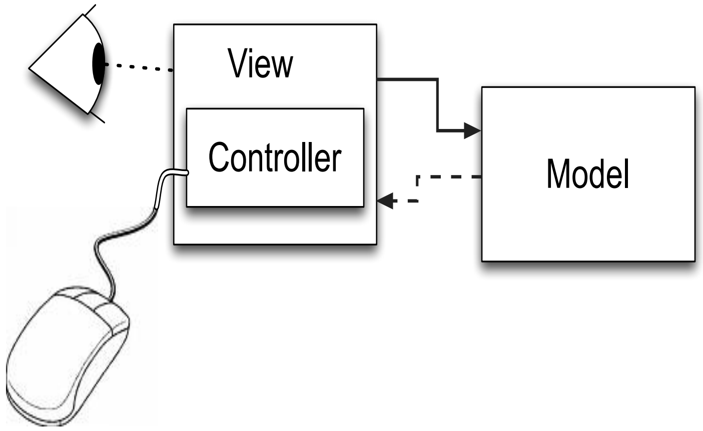
Credit: Joseph Mack for original code
http://www.austintek.com/mvc/

Theory and Practice

- **MVC in Theory**
  - View and Controller both refer to Model directly
  - Model uses the observer design pattern to inform view of changes



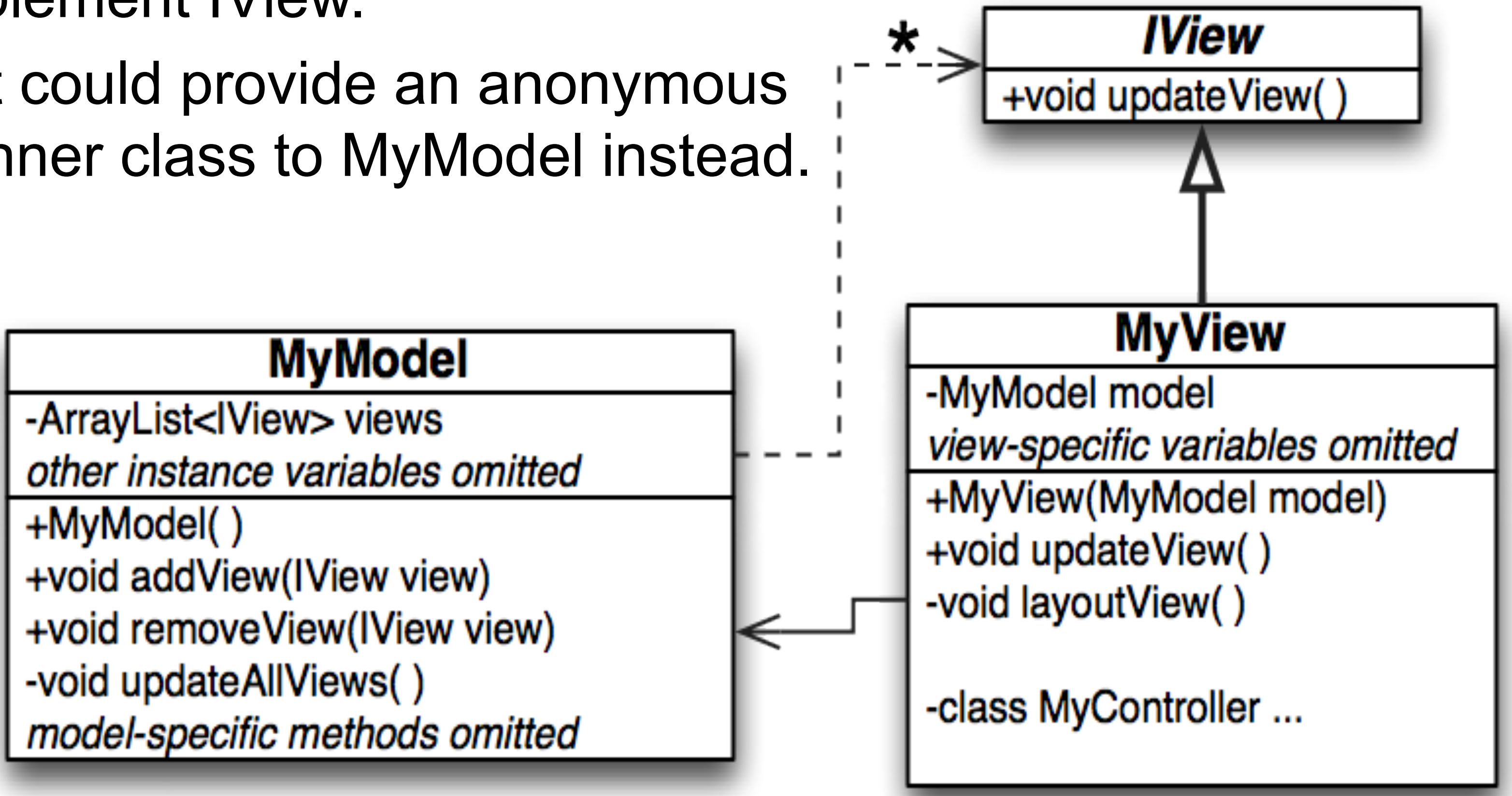
- **MVC in Practice**
  - Model is very loosely coupled with UI using the observer pattern
  - The View and Controller are tightly coupled – why?



- If the View and Controller are tightly coupled, do we still need an iView interface?
  - Why not just have the controller just tell the view to update?

MVC as UML

- NOTE: MyView does not need to implement IView.
  - It could provide an anonymous inner class to MyModel instead.



```
class MyView … {

  model.addView(new Iview() {
    void updateView() {
      …
    }}}
```

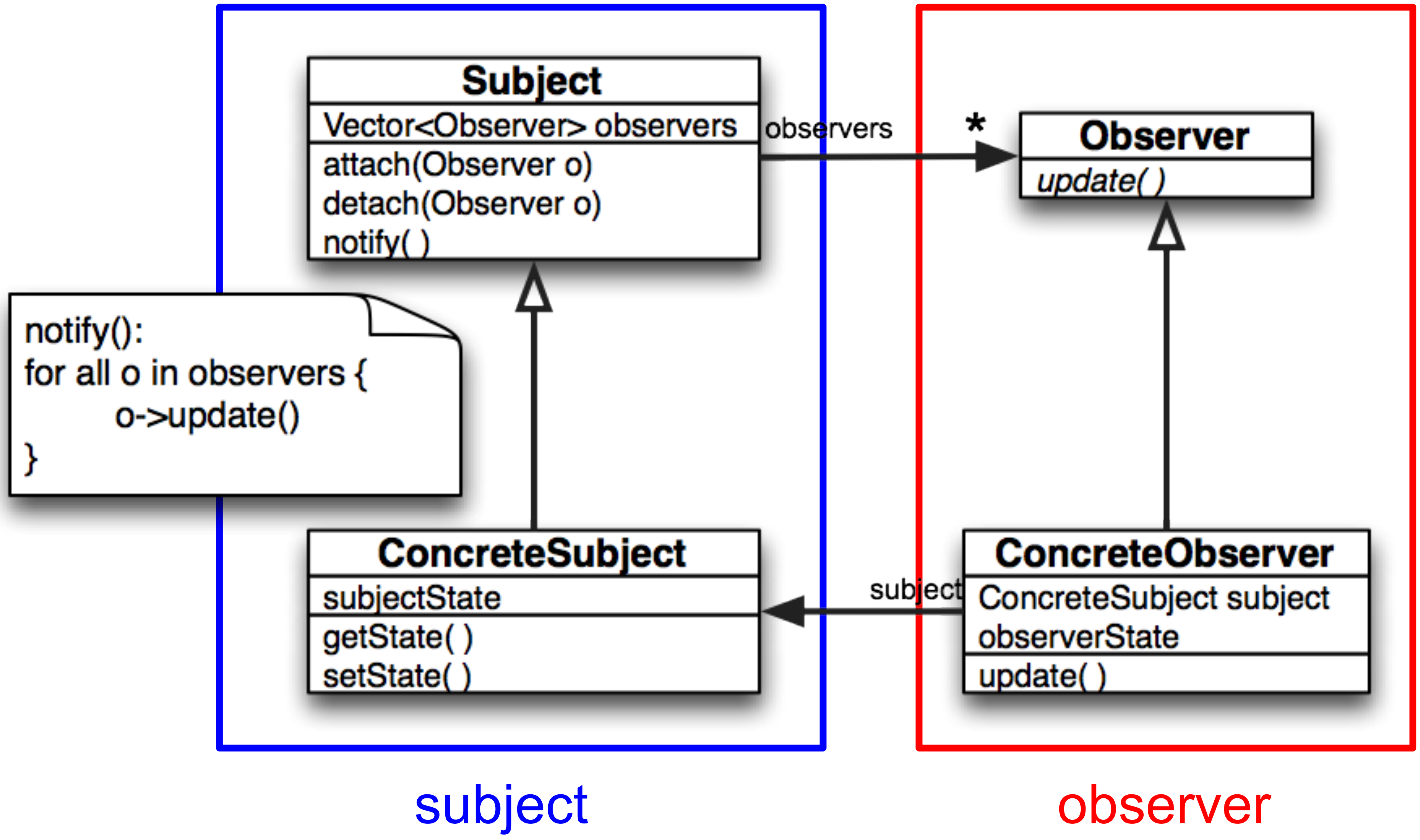
Observer Design Pattern

Observer Design Pattern

MVC is an instance of the Observer design pattern

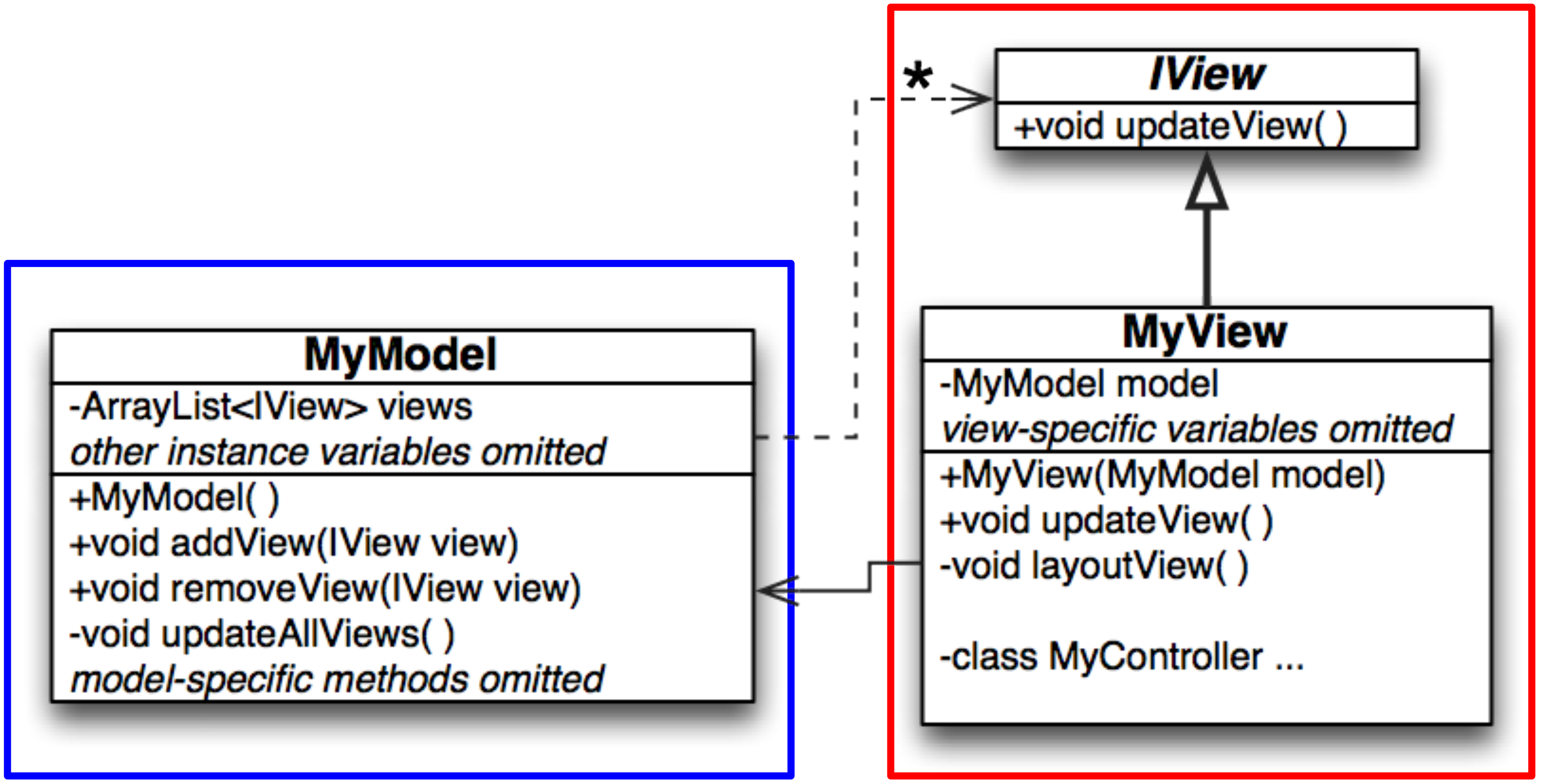
- Provides a well-defined mechanism that allows objects to communicate without knowing each others' specific types
  - Promotes loose coupling
- Related to
  - "publish-subscribe" pattern
  - "listeners"
  - delegates in C#

Observer Design Pattern



```
notify():
for all o in observers {
        o->update()
}
```

MVC as Observer Pattern

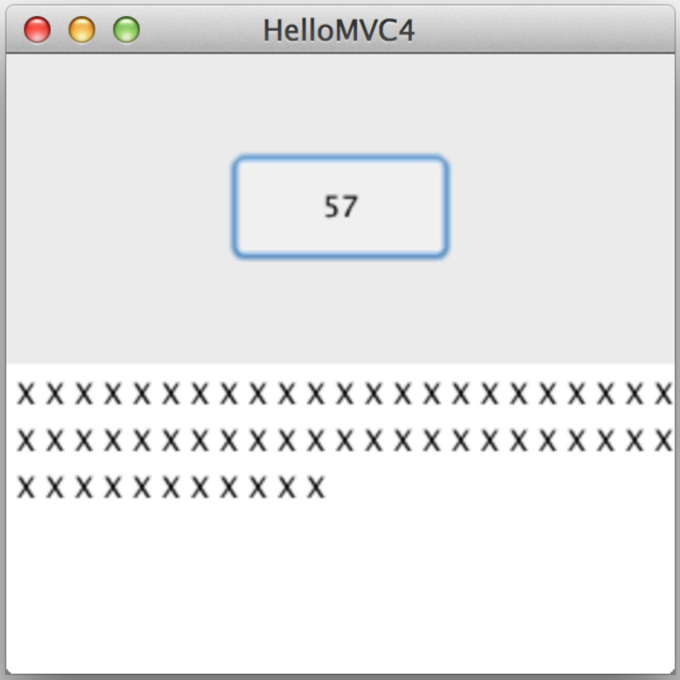


subject

observer

HelloMVC4 Code Demo

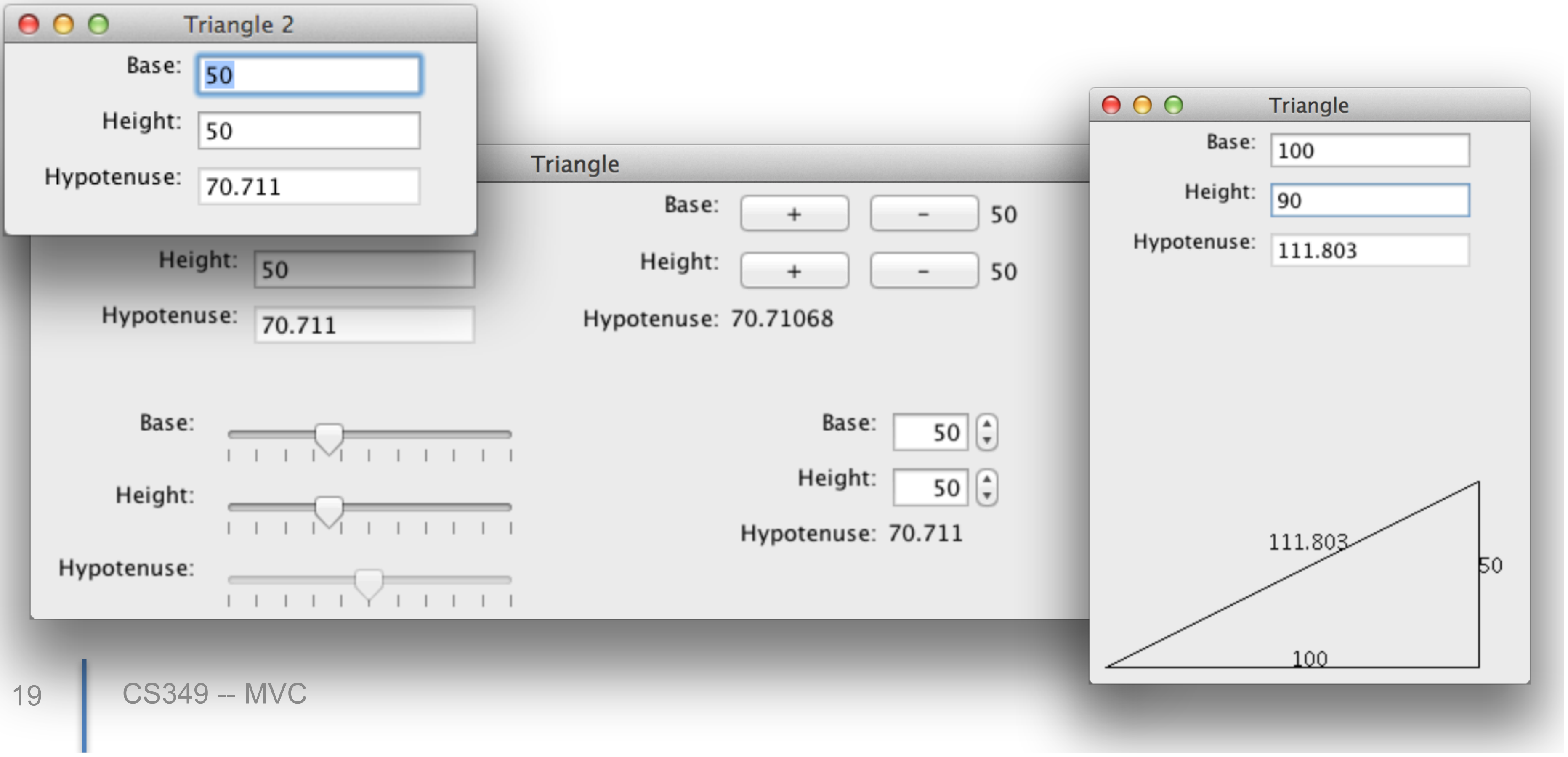
- java.util provides an Observer interface and Observable class
  - Observer is like Iview
    i.e. the View implements Observer
  - Observable is the “Subject” being observed
    i.e. the Model extends Observable
  - base class maintains a list of Observers and methods to notify them



```
X X X X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X X X X X X X X X X X X X
X X X X X X X X X X X
```

Triangle Code Demos

- Series of demo programs that use MVC
- Program requirements:
  - **vary** base and height of right triangle, display hypotenuse
- TriangleModel
  - stores base and height, calculates hypotenuse
  - constrains base and height values to acceptable range

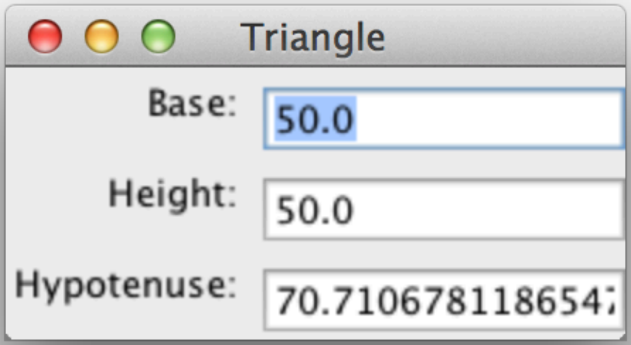


Triangle: Main_1 and Main_2

Issues with SimpleTextView

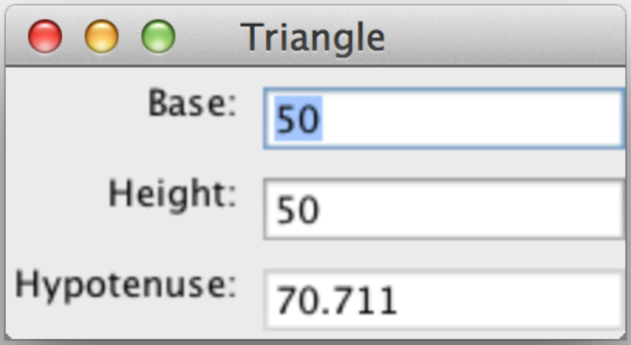
- Precision of Hypotenuse varies; sometimes wider than the textbox.
- Hypotenuse can be edited but that doesn't change the model.
- Tabbing or clicking out of base or height doesn't do anything; must hit 'Enter'.

SimpleTextView



Base: 50.0
Height: 50.0
Hypotenuse: 70.710678118654

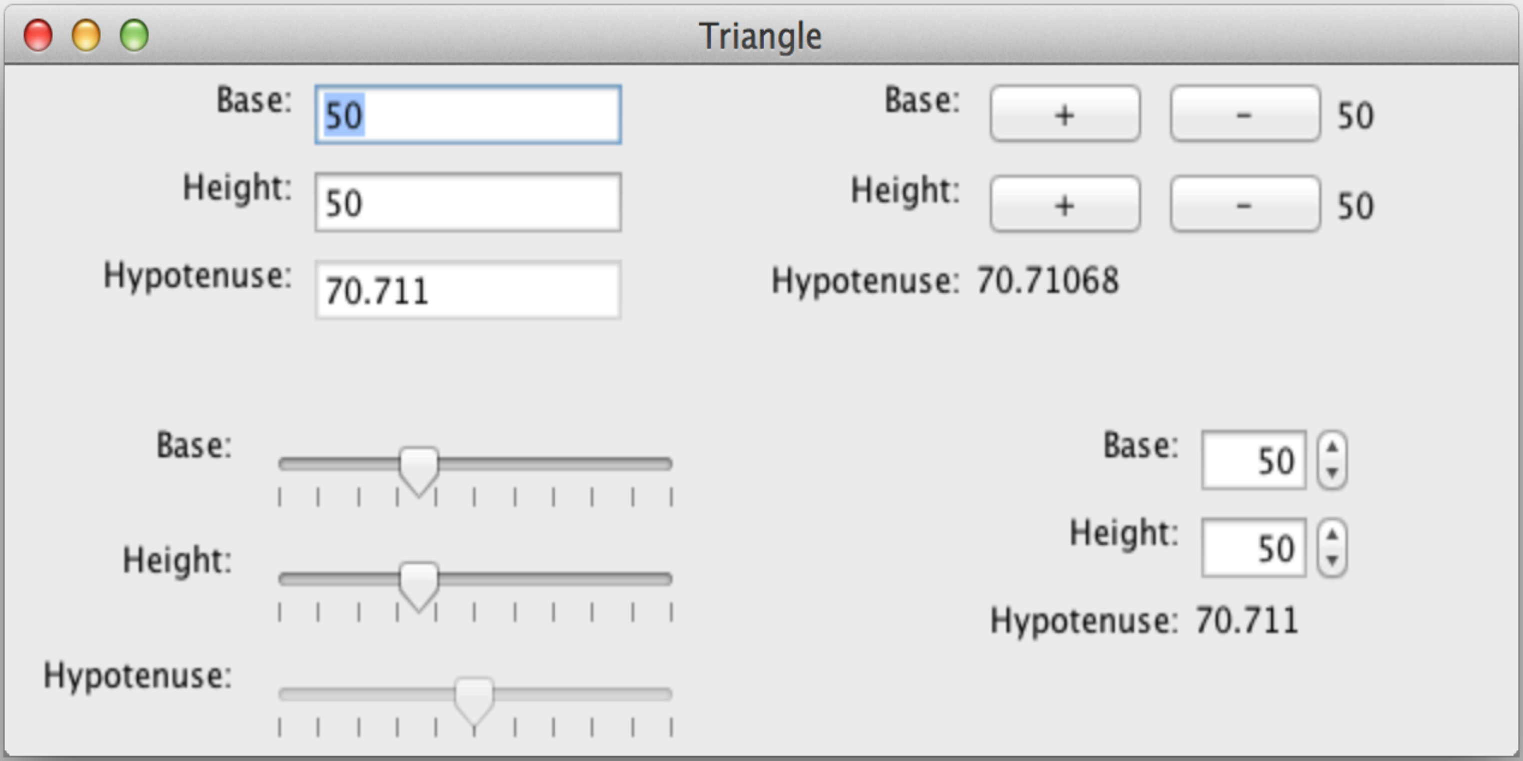
TextView



Base: 50
Height: 50
Hypotenuse: 70.711

Triangle: Main_3

Multiple Views


Triangle

Base: 50
Height: 50
Hypotenuse: 70.711

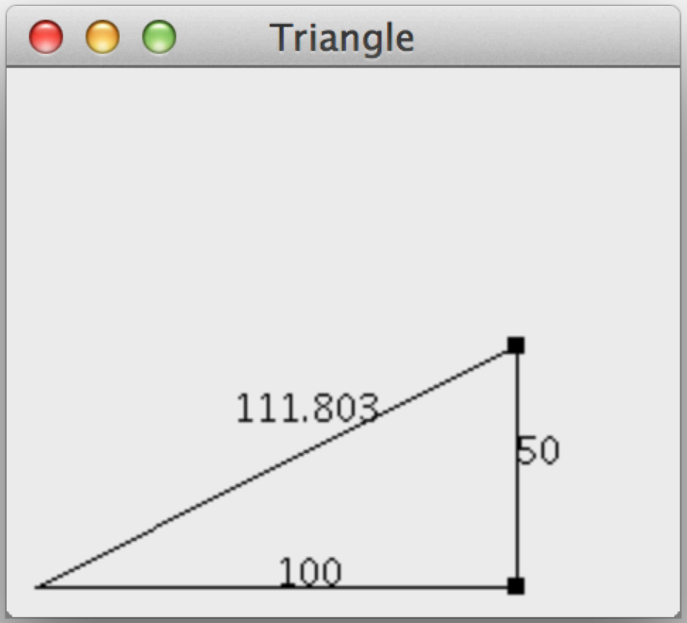
Base: + - 50
Height: + - 50
Hypotenuse: 70.71068

Base:
Height:
Hypotenuse:

Base: 50
Height: 50
Hypotenuse: 70.711

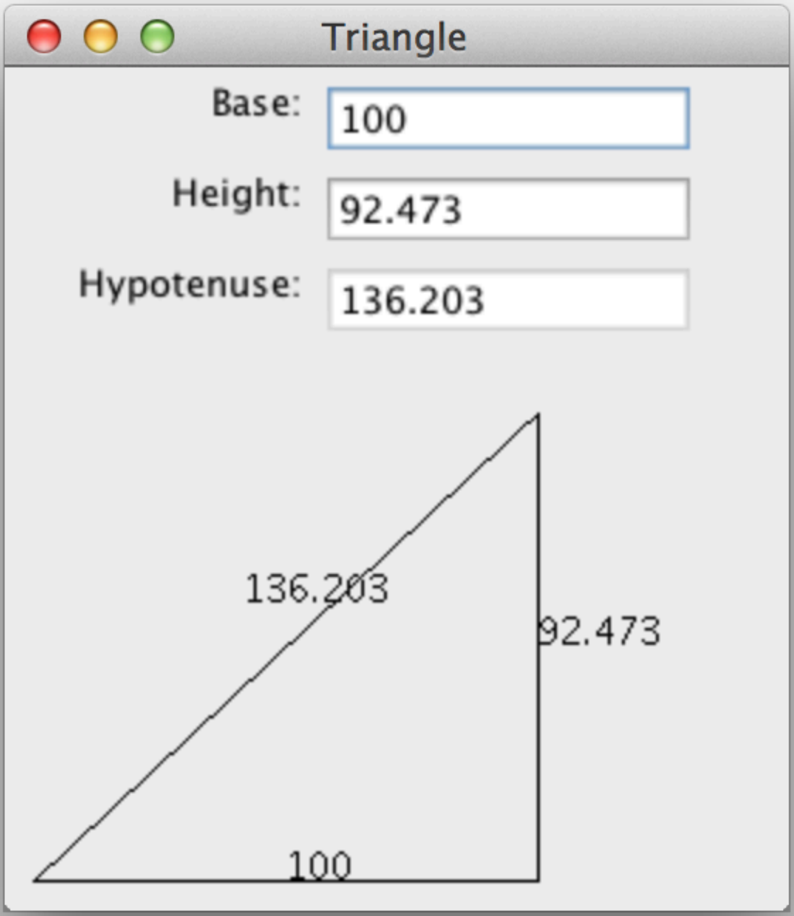
Triangle: Main_4

Graphical View



Triangle: Main_6

Graphical View



Base: 100

Height: 92.473

Hypotenuse: 136.203

Practical Details

MVC Implementation Process

- Set up the infrastructure
  - Create three or more empty classes:
    - the model
    - one or more view/controller classes (extends JComponent or JPanel)
    - a class containing the main method
  - In the main method:
    - create an instance of the model
    - create instances of the views/controllers, passing them a reference to the model
    - display the view(s) in a frame

Hello MVC 1- 4

```java
public class Main{

    public static void main(String[] args){
        JFrame frame = new JFrame("HelloMVC1");

        // create Model and initialize it
        Model model = new Model();
        // create Controller, tell it about model
        Controller controller = new Controller(model);
        // create View, tell it about model and controller
        View view = new View(model, controller);
        // tell Model about View.
        model.setView(view);
```

```java
class View extends JPanel implements IView {

    // the view's main user interface
    private JButton button;

    // the model that this view is showing
```

hellomvc1 /
main.java
view.java

MVC Implementation Process (cont.)

- Build and test the model
  - Design, implement, and test the model
    - add commands used by controllers to change the model
    - add queries used by the view to update the display
  - Call `updateAllViews()` just before exiting any public method that changes the model's data
- Build the Views and Controllers
  - Design the UI as one or more views. For each view:
    - Construct widgets
    - Lay the widgets out in the view
    - Write and register appropriate controllers for each widget
    - Write updateView() to get and display info from the model
    - Register view (with updateView() method) with the model

Summary

MVC Rationale 1: Change the UI

- Separation of concerns enables alternative forms of interaction with the same underlying data.
  - Data and how it is manipulated (the model) will remain fairly constant over time.
  - How we present and manipulate that data (view and controller) via the user interface will likely change more often than the underlying model.
  - E.g. transitioning an application from desktop to smartphone to watch versions.

MVC Rationale 2: Multiple Views

- Separation of concerns enables <u>multiple, simultaneous views</u> of the data.
- Given the same set of data, we may want to render it in multiple ways:
  – a table of numbers
  – a pie chart
  – a line graph
  – an audio stream
  – ...
- A separate model makes it easier for different UI components to use the same data
  – Each view is unencumbered by the details of the other views
  – Reduces dependencies on the GUI that could change

MVC Rationale 3: Testing

- Separation of concerns enables one to more easily develop and test data-specific manipulations that are independent of the user interface
  - Build tests that exercise the model independent of the interface
  - Makes automated tested of user interfaces practical