

Model View Controller (MVC)

Multi-window support

Benefits of MVC

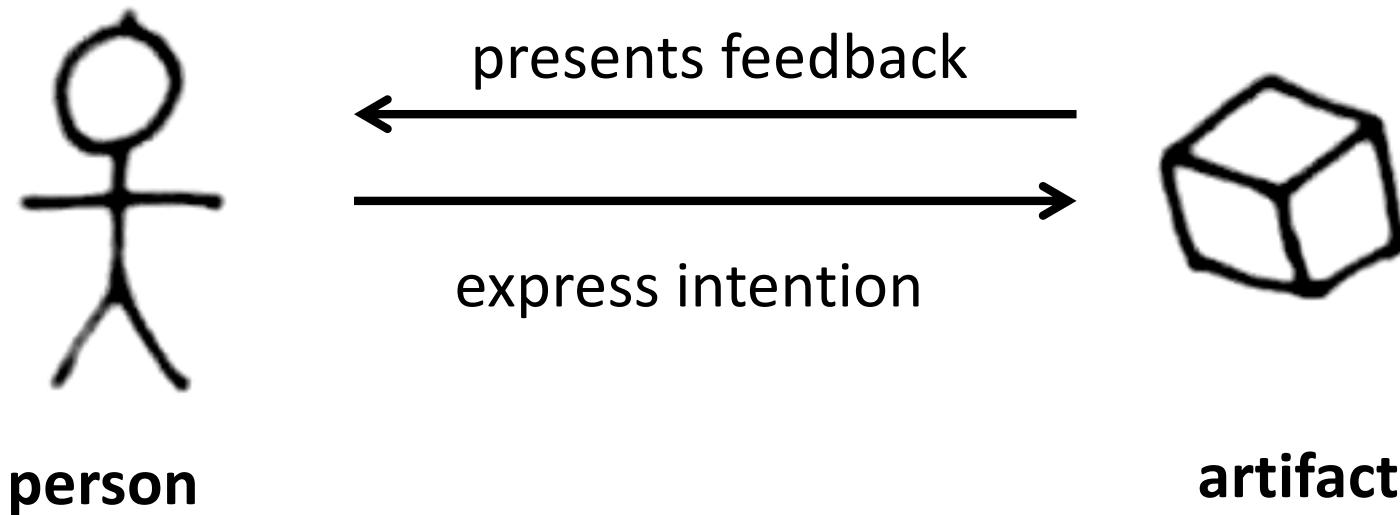
Java Implementation

Widgets

Recall: User Interaction

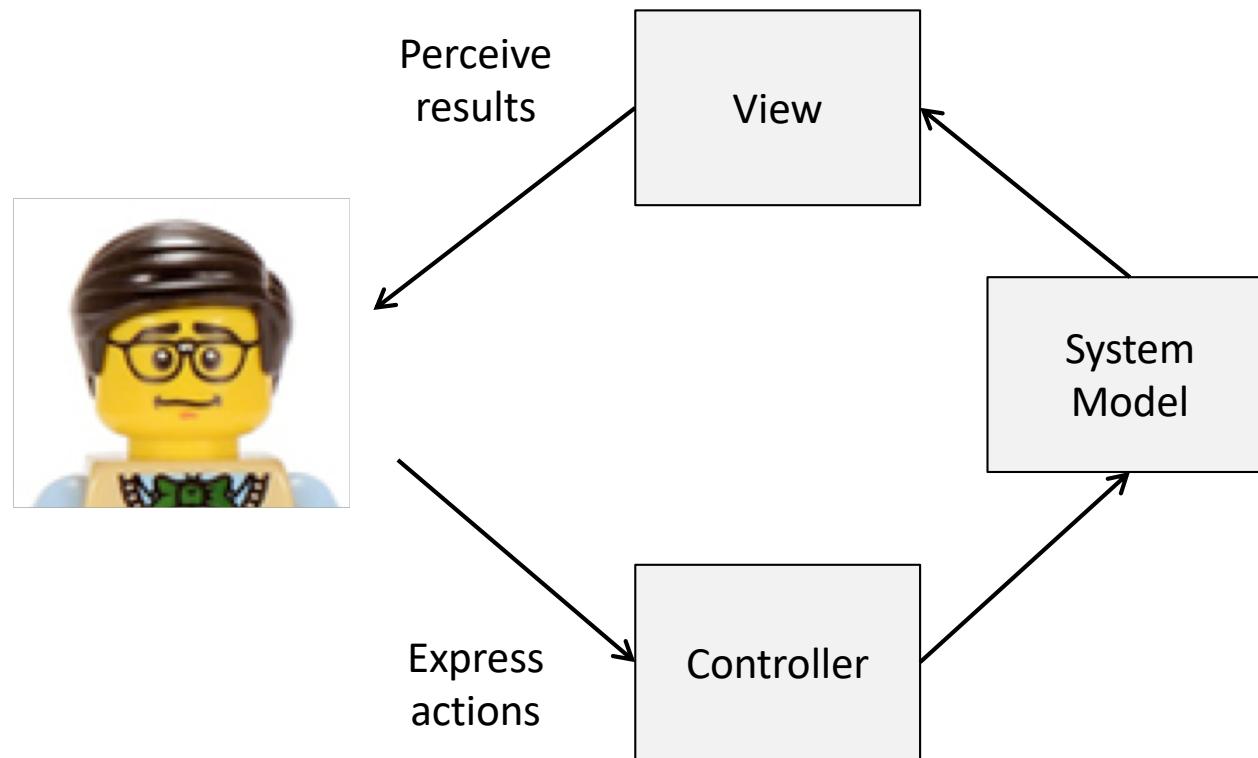
People typically interact with technology to perform a task.

They determine what they want to do, and provide guidance to the system, which gives them feedback.



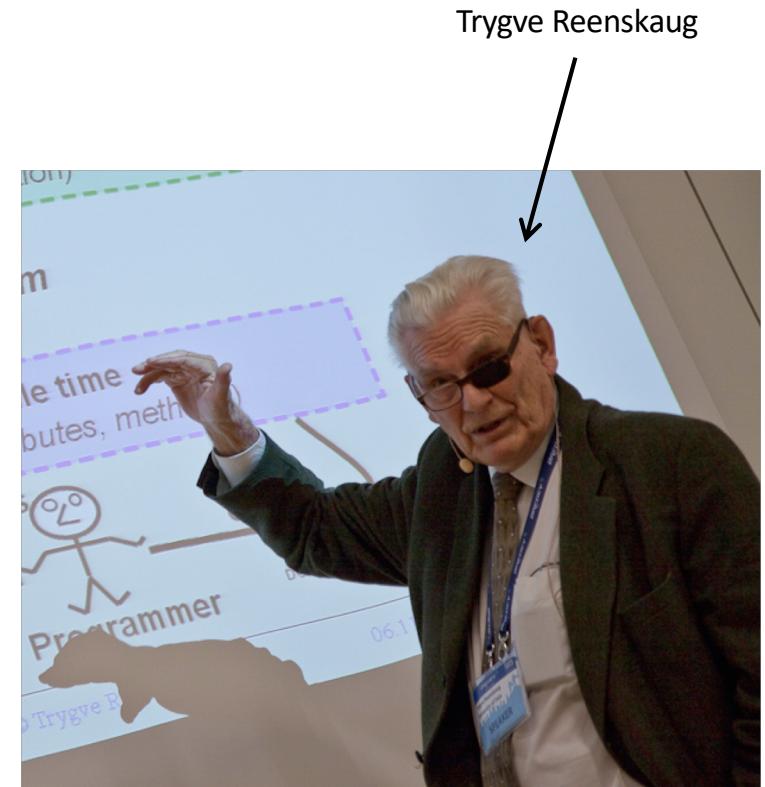
User Interaction

We often design our systems to directly support this style of interaction.
This architectural pattern is called **Model-View-Controller**.



Model-View-Controller (MVC)

- Developed at Xerox PARC in 1979 by Trygve Reenskaug
 - for Smalltalk-80 language, the precursor to Java
- Standard design pattern for GUIs
- Used at many levels
 - Overall application design
 - Individual components
- Many variations of MVC idea:
 - Model-View-Presenter
 - Model-View-Adapter
 - Hierarchical Model-View-Controller



Why use MVC?

1. Clear separation of “business logic” and the user-interface logic.

This means that the View (output) and Controller (input) can be changed without changing the underlying Model.

- When is this useful?

- Adding support for a new input device (e.g. voice control, touchscreen)
 - Adding support for a new type of output device (e.g. different size screen)

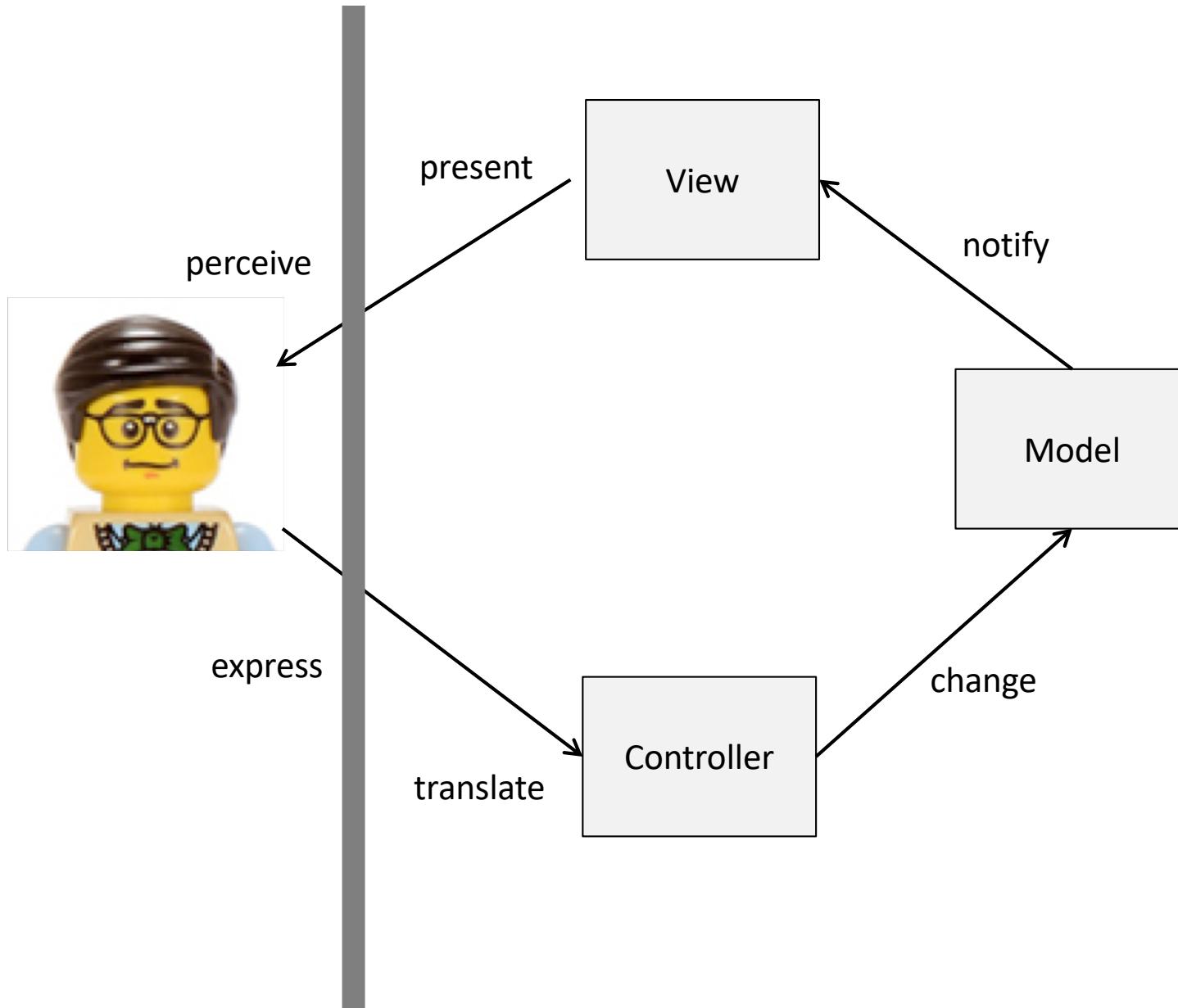
2. Supports multiple views of the underlying data/model.

- When is this useful?

- Viewing numeric data as a table, a line graph, a pie chart, ...
 - Displaying simultaneous “overview” and “detail” views
 - Enabling “edit” and “preview” views

3. Separation of concerns in code (code reuse, ease of testing)

Model-View-Controller (MVC)



How can we design PowerPoint?

The screenshot shows a Microsoft PowerPoint slide titled "Model-View-Controller (MVC)". The slide is part of a presentation titled "1.1 Introduction". The ribbon menu is visible at the top, showing "Home" as the active tab. The left sidebar contains thumbnails for slides 14 through 18. Slide 16 is highlighted with a red border.

Model-View-Controller (MVC)

```
graph TD; User((User)) -- "perceive" --> View[View]; View -- "present" --> User; View -- "notify" --> Model[Model]; Model -- "change" --> Controller[Controller]; Controller -- "translate" --> User;
```

The diagram illustrates the MVC architecture. It features a central vertical bar. On the left, a user's "mental model" is shown in a cloud, with arrows labeled "perceive" pointing from the user to a "View" box and "express" pointing from the "View" box back to the user. On the right, a "Model" box is connected to a "Controller" box with an arrow labeled "change". The "Controller" box has an arrow labeled "translate" pointing back to the user's "mental model". A "system model" is also shown in a cloud near the "Model" box.

Click to add notes

Slide 16 of 35

Notes Comments

7

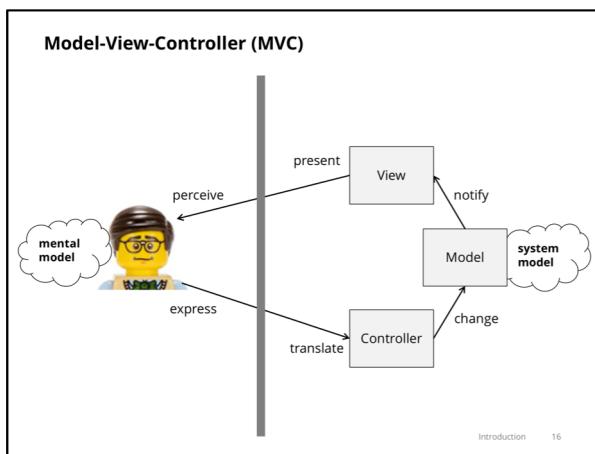
Multiple Views

- Many applications have multiple views of one “document”

This screenshot shows a Microsoft PowerPoint slide titled "Model-View-Controller (MVC)". The slide contains a diagram illustrating the MVC architecture. At the top left, there is a list of slide titles and numbers. The main content features a central diagram with four components: "View", "Model", "Controller", and "mental model". Arrows show the flow of information: "View" presents to "Controller", "Controller" translates to "View", "Controller" notifies "Model", and "Model" changes. There are also arrows from "View" to "mental model" labeled "perceive" and from "mental model" to "Controller" labeled "express". The slide is numbered 16 and is part of a presentation titled "1.1 Introduction".

This screenshot shows a Microsoft PowerPoint slide titled "Model-View-Controller (MVC)". The slide contains a diagram illustrating the MVC architecture. The layout is identical to the first slide, featuring a central diagram with components "View", "Model", "Controller", and "mental model". Arrows indicate the flow of information between these components. The slide is numbered 16 and is part of a presentation titled "1.1 Introduction".

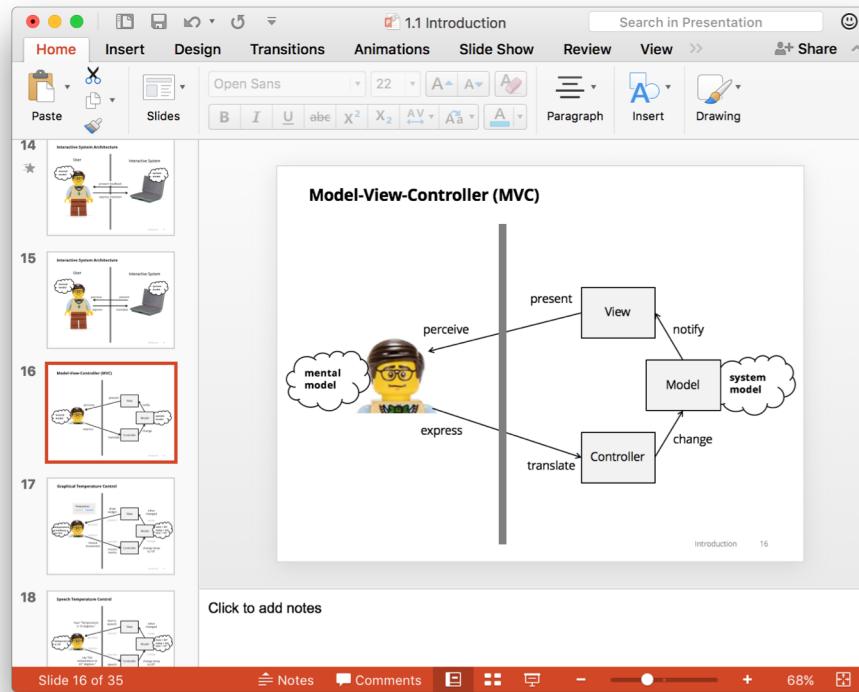
This screenshot shows a Microsoft PowerPoint slide displaying a grid of 25 thumbnail previews of other slides. The thumbnails include various diagrams and text snippets related to the MVC architecture and user interfaces. The slide is numbered 16 and is part of a presentation titled "1.1 Introduction".



This screenshot shows a Microsoft PowerPoint slide titled "Model-View-Controller (MVC)". The slide contains a diagram illustrating the MVC architecture. The layout is identical to the previous diagrams, featuring a central diagram with components "View", "Model", "Controller", and "mental model". Arrows indicate the flow of information between these components. The slide is numbered 16 and is part of a presentation titled "1.1 Introduction".

Multiple views observations

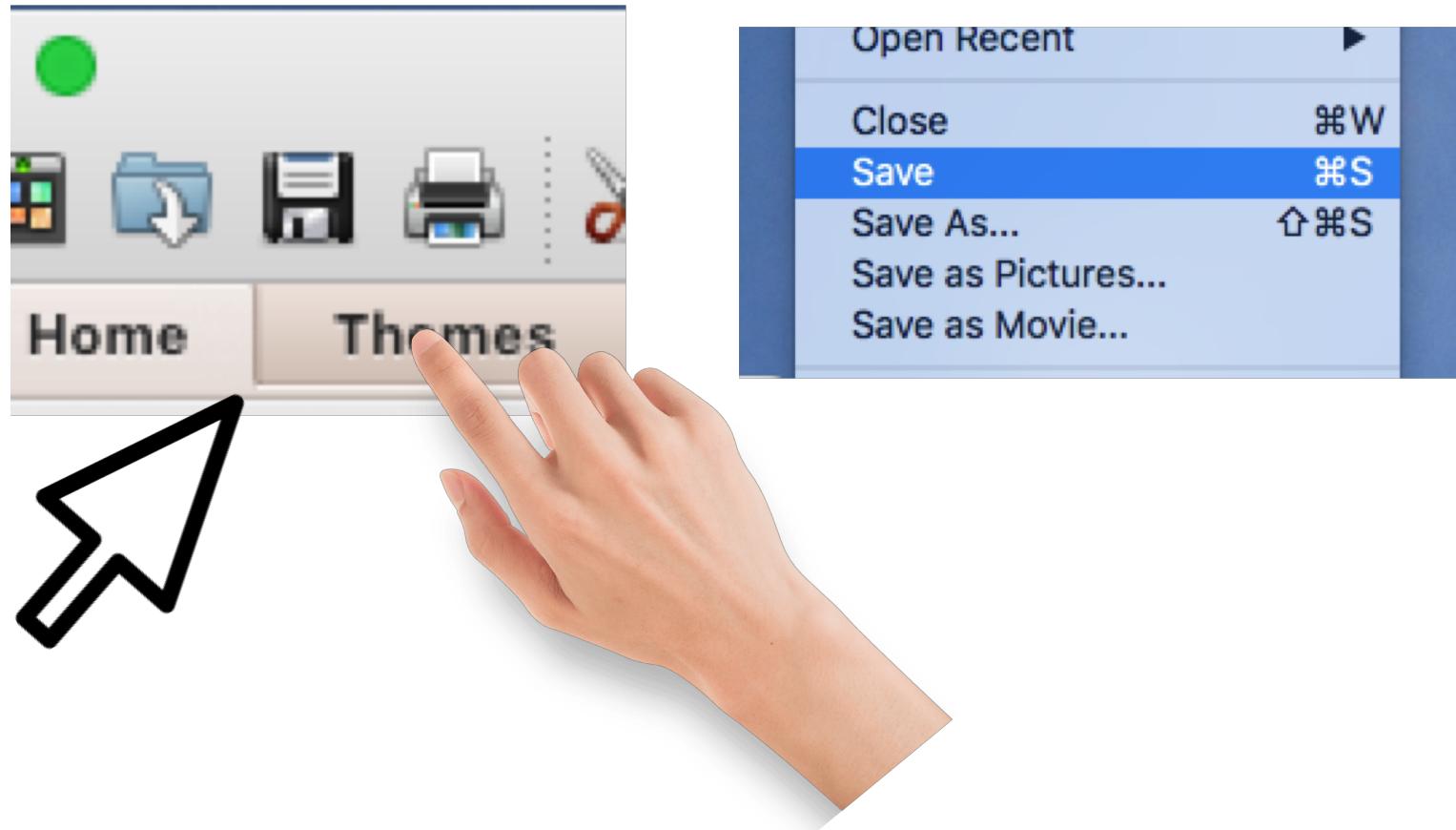
When one view changes, the others should change as well.



- How do we design software to support these observations?
 - We define a single model, that can send information to multiple views.

Multiple input mechanism for same functionality

- What if we want to add new input mechanism?
- Replaceable controllers make this possible.

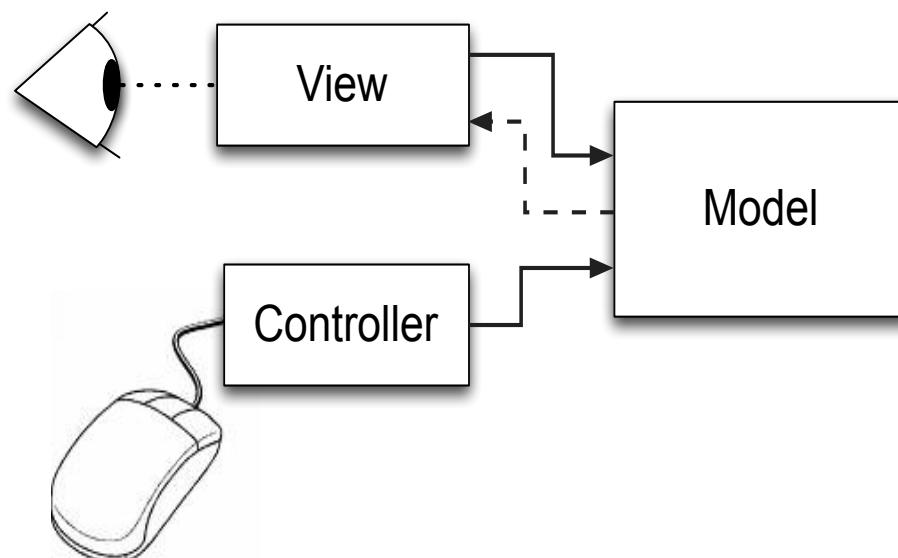


MVC Implementation

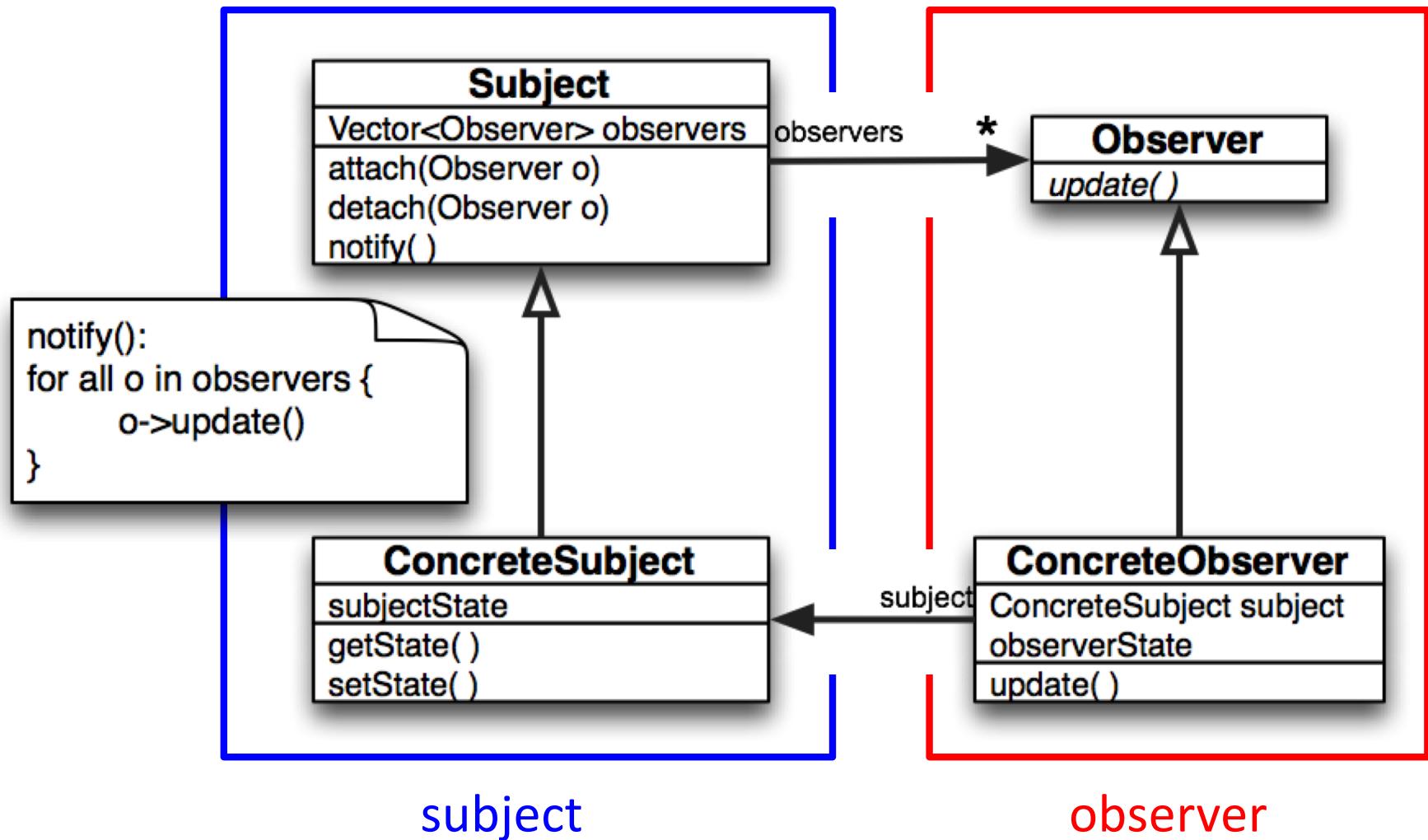
MVC is an instance of the Observer design pattern (sim. “publish-subscribe”)

In MVC, interface architecture is decomposed into three parts:

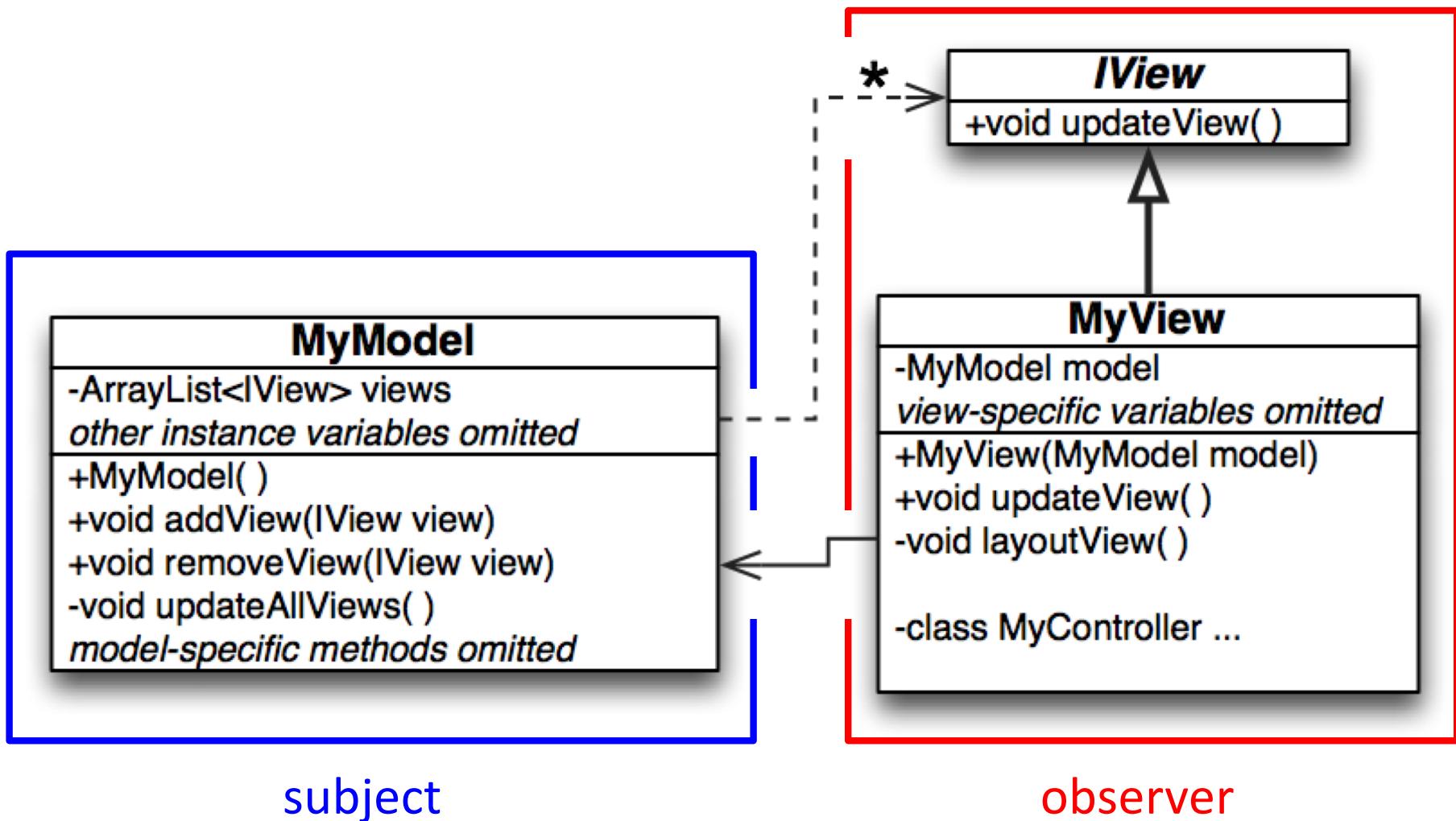
- **Model**: manages application data and its modification
- **View**: manages interface to present data
- **Controller**: manages interaction to modify data



Observer Design Pattern



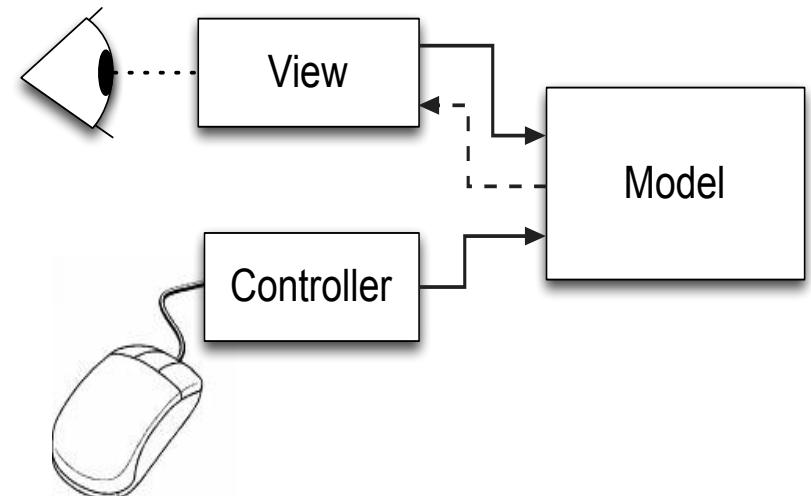
MVC as Observer Pattern



Theory and Practice

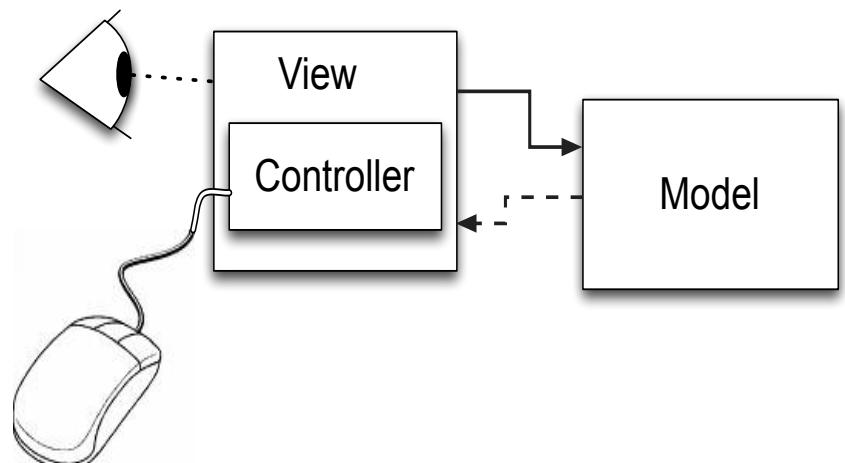
- **MVC in Theory**

- View and Controller are separate and loosely uncoupled



- **MVC in Practice**

- The View and Controller are tightly coupled. Why?



Problems with MVC

There have been issues identified with MVC since it was introduced.

1. The View should handle both input and output
 - We input and view data on the same screen, so a separate controller class makes little sense.
 - There is often view-level data manipulations that don't make sense to send to the Model. e.g. checking the format of a phone number, highlighting an invalid entry.
 - e.g. Model-View, Model-View-Adapter eliminate Controller class
2. One common Model doesn't always make sense
 - We often have data/state that is specific to a View. e.g. different logic on a screen based on region or language.
 - e.g. Model-View-Adapter, Model-View-Presenter mediate between Model and View, and provide additional logic.

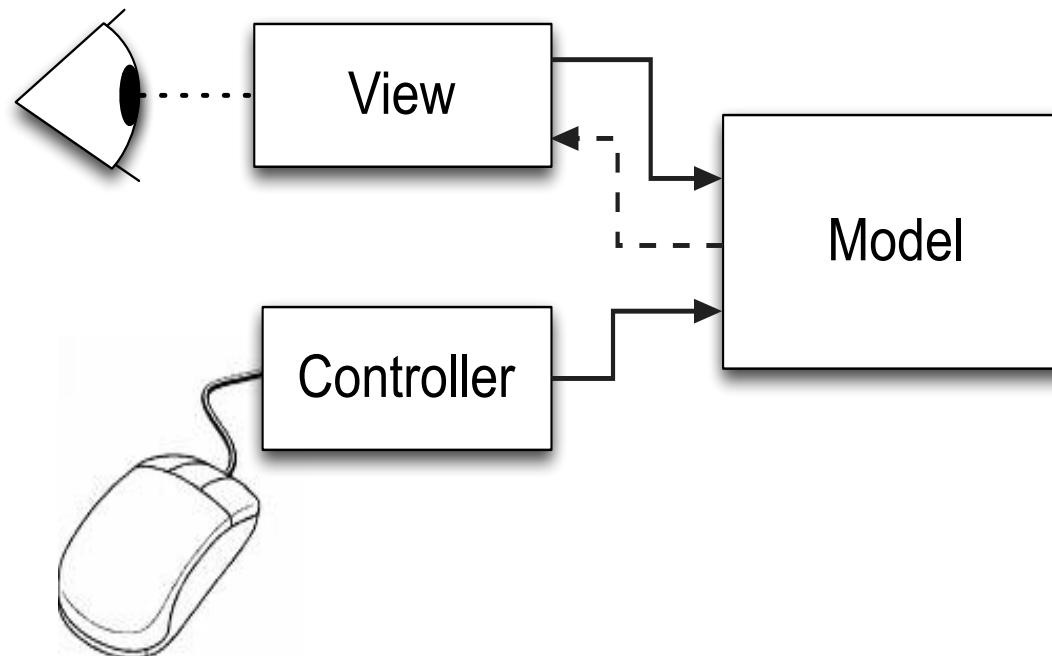
Implementing MVC in Java

Class design

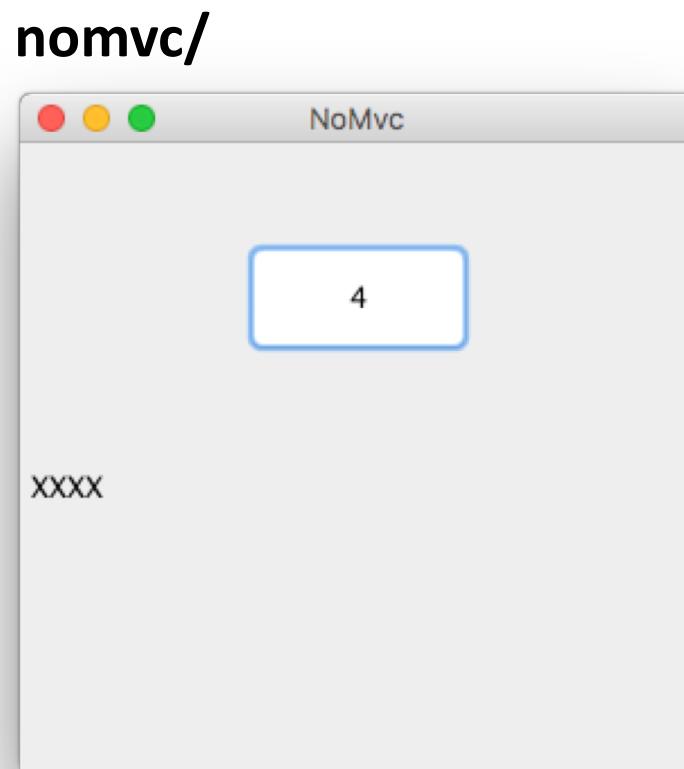
Examples

MVC Implementation

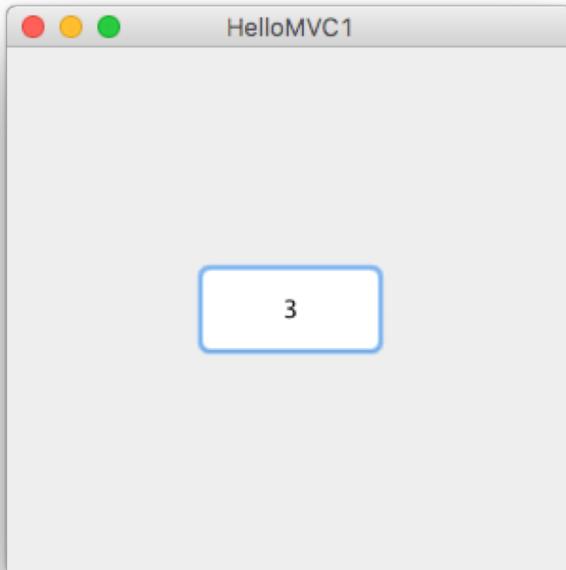
- Interface architecture decomposed into three parts:
 - **Model:** manages application data and its modification
 - **View:** manages interface to present data
 - **Controller:** manages interaction to modify data



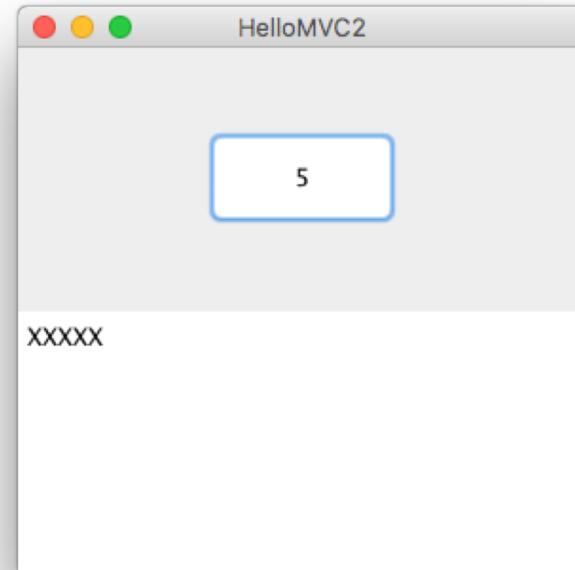
No MVC: Motivating Example



hellomvc1/
1 view



hellomvc2/
2 (or more) views



inspired by Joseph Mack: <http://www.austintek.com/mvc/>
- (also a good MVC explanation, shows how to use Java Observer class)

View Class Outline

```
class View implements IView {  
    private Model model; // the model this view presents  
  
    View(Model model, Controller controller) {  
  
        ... create the view UI using widgets ...  
  
        this.model = model; // set the model  
        // setup the event to go to the controller  
        widget1.addListener(controller);  
        widget2.addListener(controller);  
    }  
  
    public void updateView() {  
        // update view widgets using values from the model  
        widget1.setProperty(model.getValue1());  
        widget2.setProperty(model.getValue2());  
        ...  
    }  
}
```

Controller Class Outline

```
class Controller implements Listener {  
    Model model; // the model this controller changes  
  
    Controller(Model model) {  
        this.model = model; // set the model  
    }  
  
    // events from the view's widgets  
    // (often separated to 1 method per widget)  
    public void actionPerformed(Event e){  
        // note the controller does need to know about view  
        if (widget1 sent event)  
            model.setValue1();  
        else if (widget2 sent event)  
            model.setValue2();  
        ...  
    }  
}
```

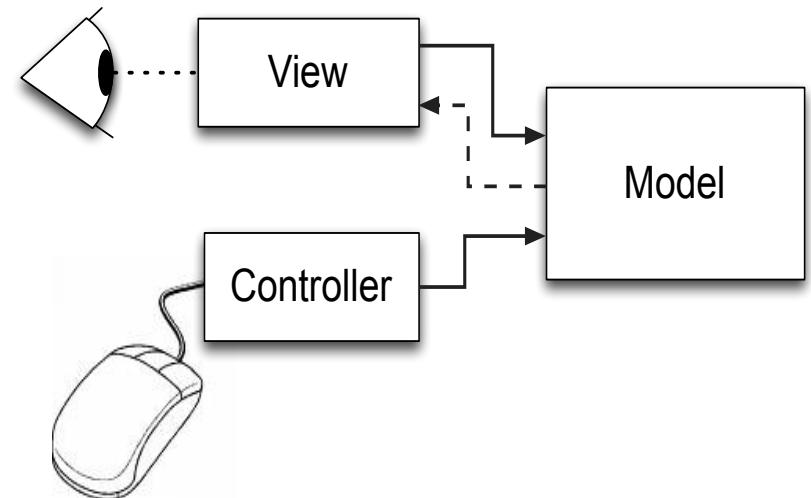
Model Class Outline

```
class Model {  
    List<IView> views; // multiple views  
    public void addView(IView view) {...} // add view observer  
  
    // get model values  
    public type getModelValue1() { return value1; }  
    public type getModelValue2() { return value2; }  
    ... more value getters ...  
  
    // set model values  
    public void setModelValue1(type value) {  
        value1 = value; notifyObservers();  
    }  
  
    ... more setters, each calls notifyObservers() ...  
  
    // notify all IView observers  
    private void notifyObservers() {  
        for (IView view: views) view.updateView();  
    }  
}
```

Theory and Practice

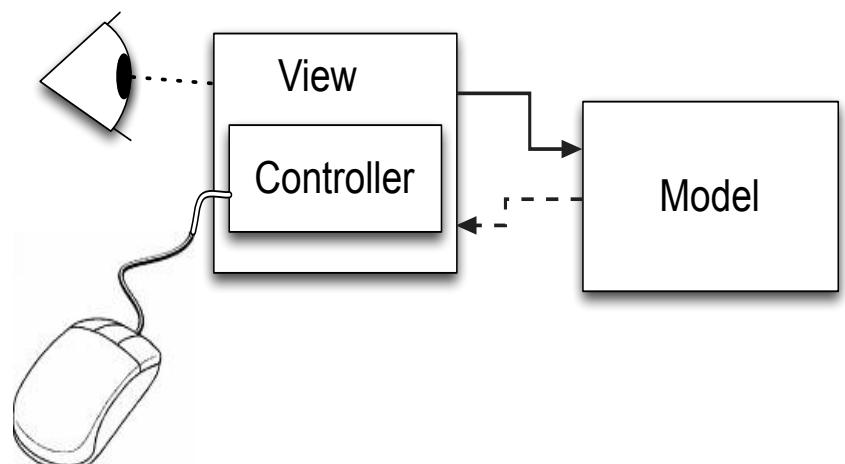
- **MVC in Theory**

- View and Controller are separate and loosely uncoupled



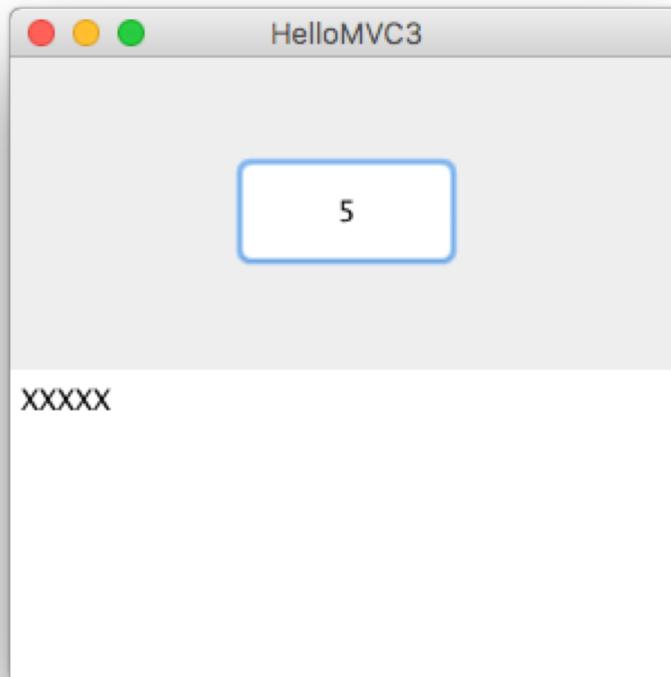
- **MVC in Practice**

- The View and Controller are tightly coupled. Why?



Why doesn't the Controller just directly tell the View to update?
Why "go through" the Model?

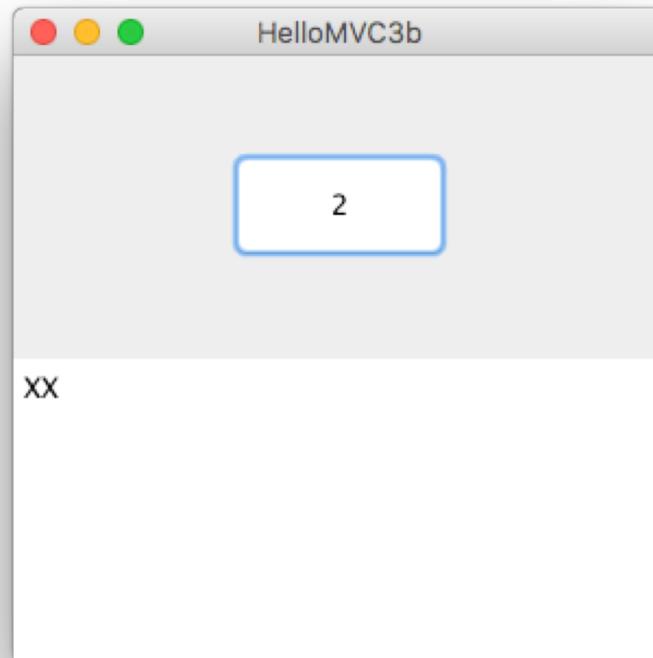
hellomvc3/ Controller code in View



Model Updates Can be Considered an Event

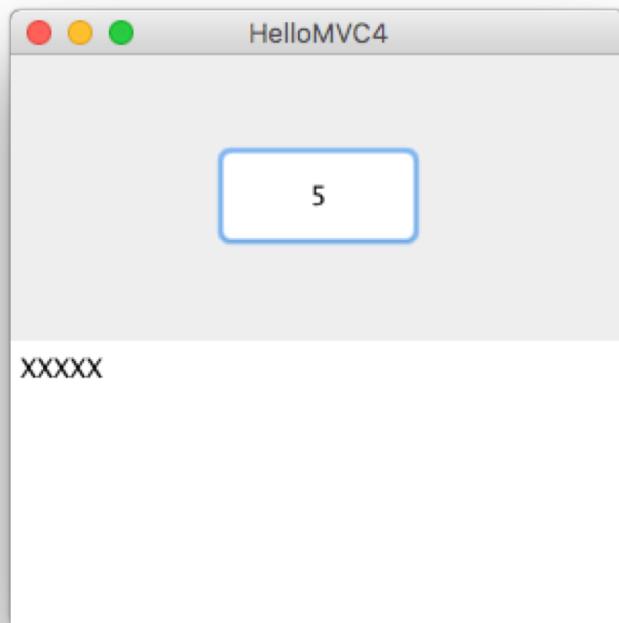
- View can use an anonymous inner class of type IView to create a “ModelListener” for Model to register

hellomvc3b/
anonymous ModelListener



hellomvc4/

- java.util provides 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 has list of Observers and method to notify them

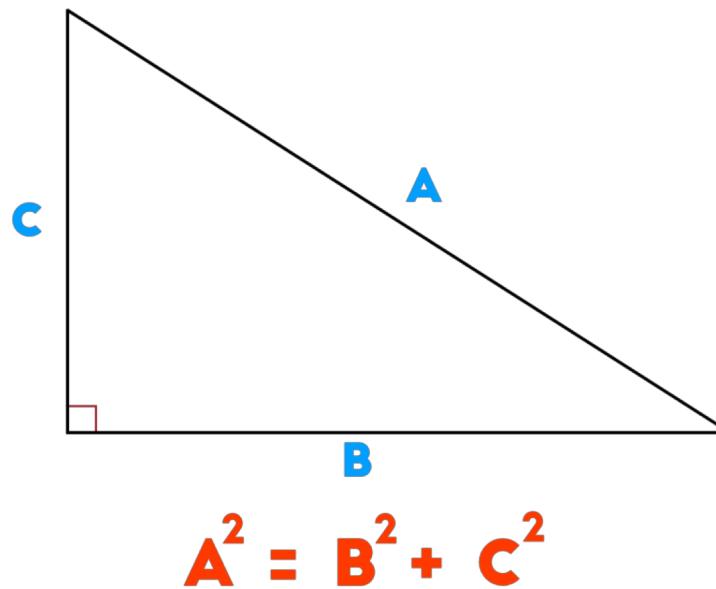


Optimizing View Updates

- Each viewUpdate, *everything* in *every view* is refreshed from model
- Could add parameters to viewUpdate to indicate *what* changed
 - if view knows it isn't affected by change, can ignore it
- But, **simpler is better**
 - early optimization only introduces extra complexity that causes bugs and adds development time
 - don't worry about efficiency until you have to:
just update the entire interface

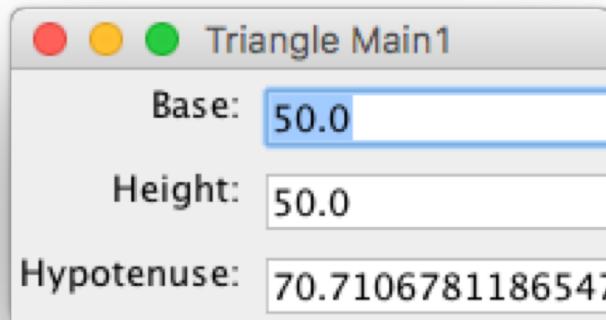
Triangle MVC Code Demos

- Program requirements:
 - vary base and height of right triangle, display hypotenuse
- TriangleModel.java
 - stores base and height, calculates hypotenuse
 - constrains base and height values to acceptable range

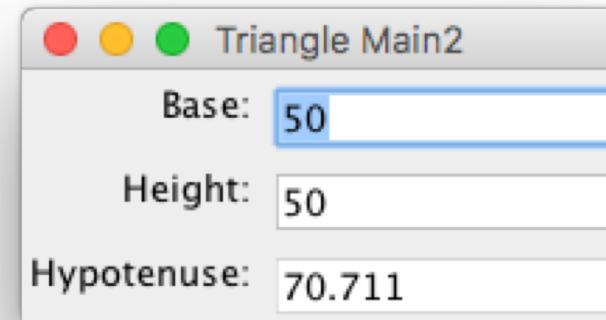


Triangle: Main1.java and Main2.java

SimpleTextView



TextView



Triangle: Main3.java

Combines Multiple Views using GridLayout

TextView

Base:

Height:

Hypotenuse:

ButtonView

Base: 50

Height: 50

Hypotenuse: 70.71068

Base:

Height:

Hypotenuse:

SliderView

Base: ^

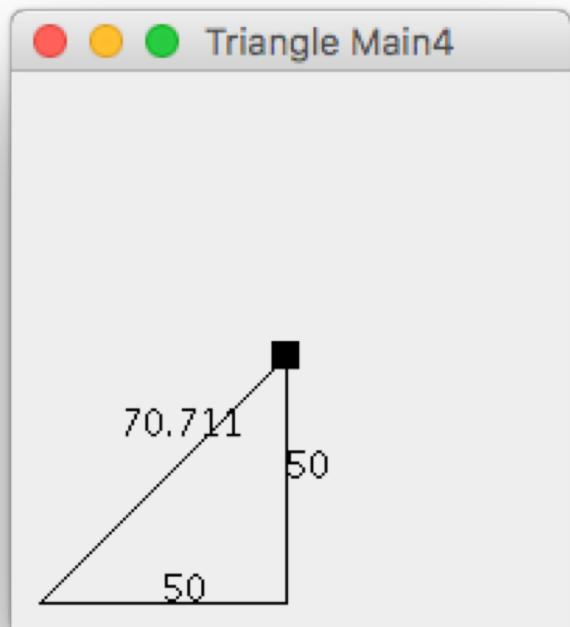
Height: ^

Hypotenuse: 70.711

SpinnerView

Triangle: Main4.java

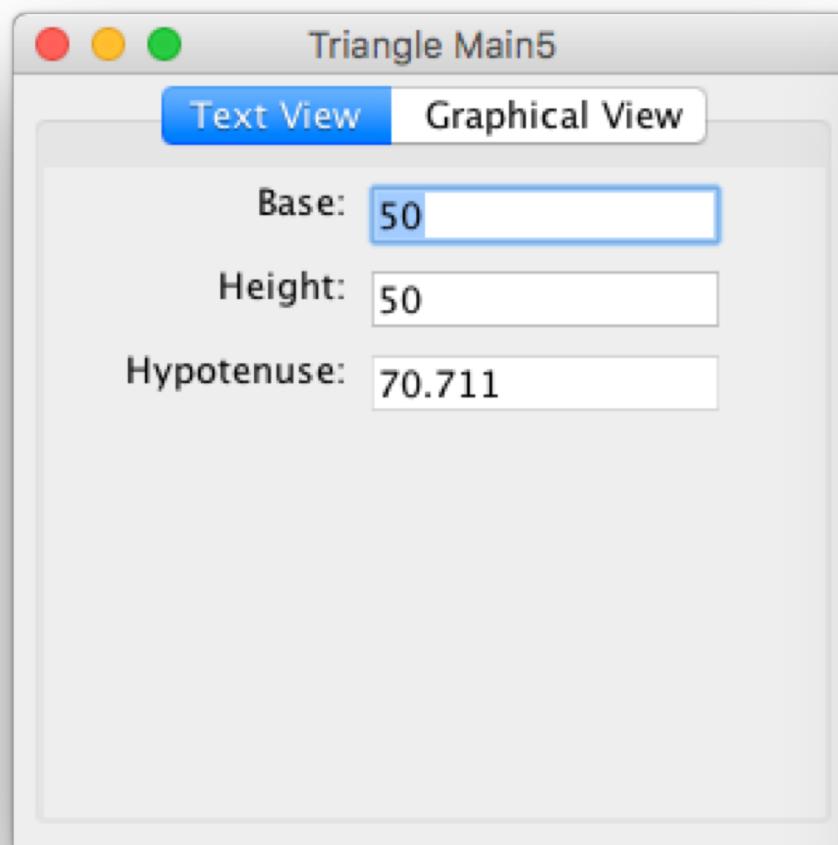
GraphicalView



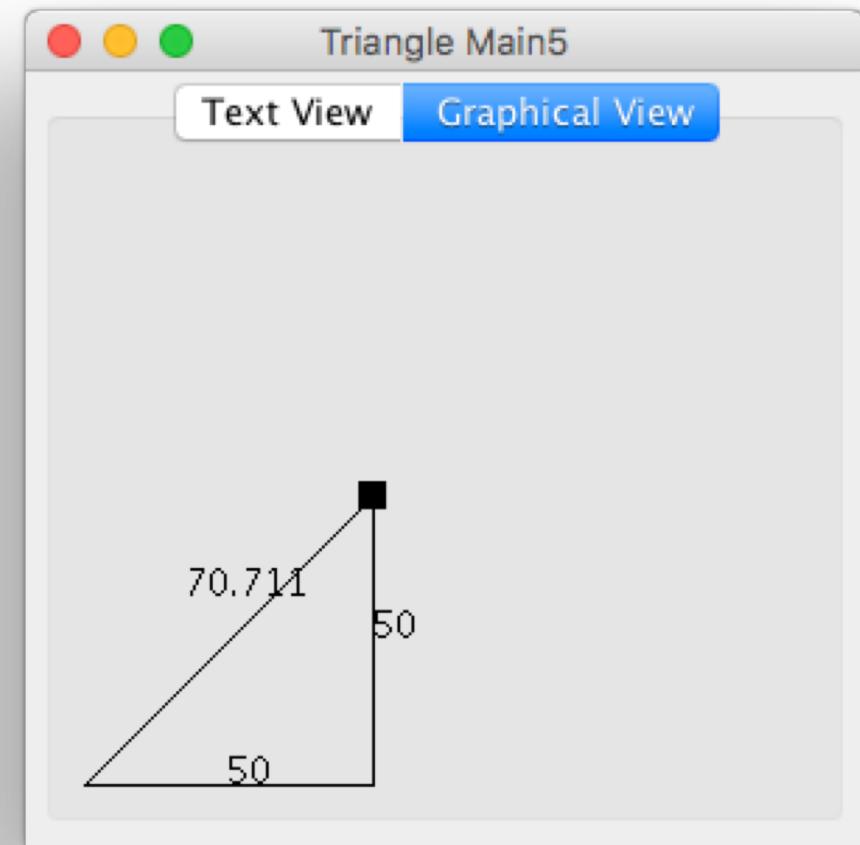
Triangle: Main5.java

Combines Multiple Views using Tab Panel

TextView



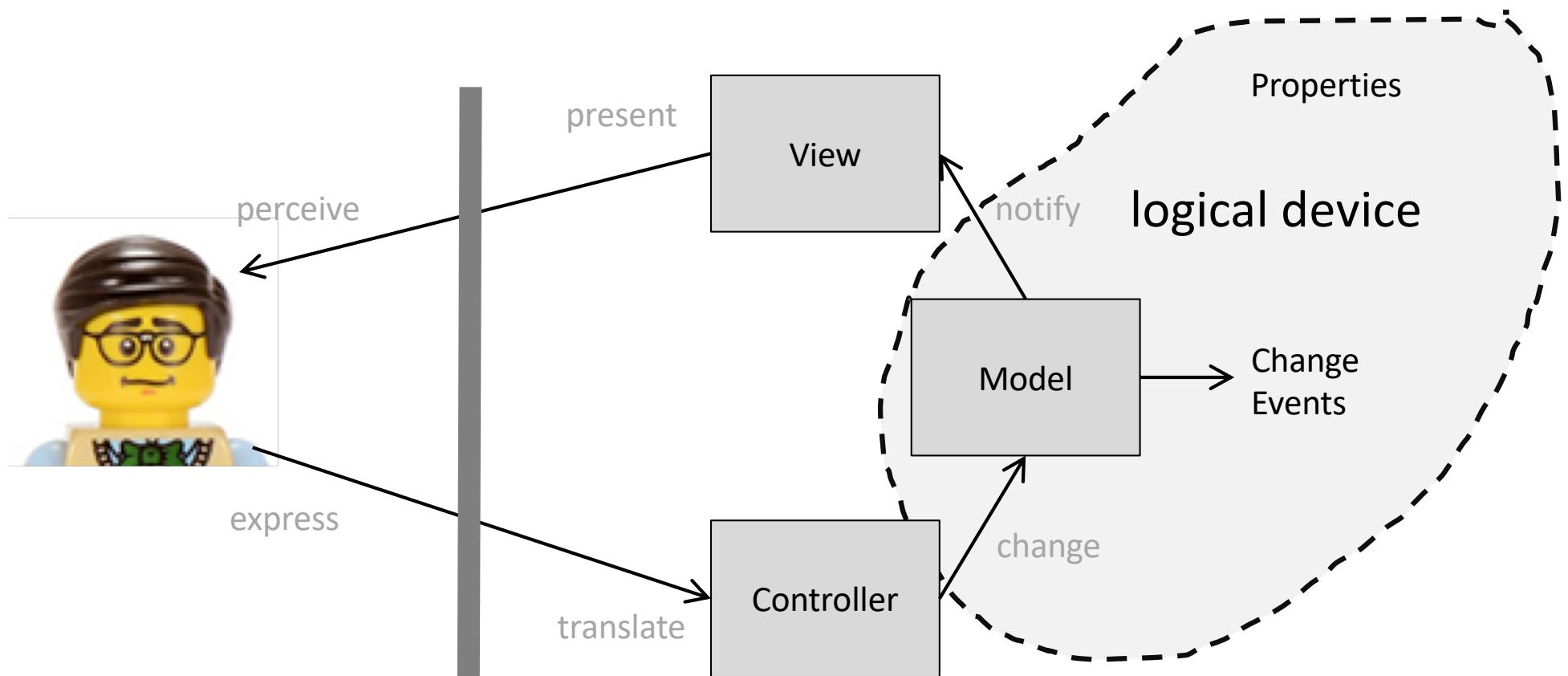
GraphicalView



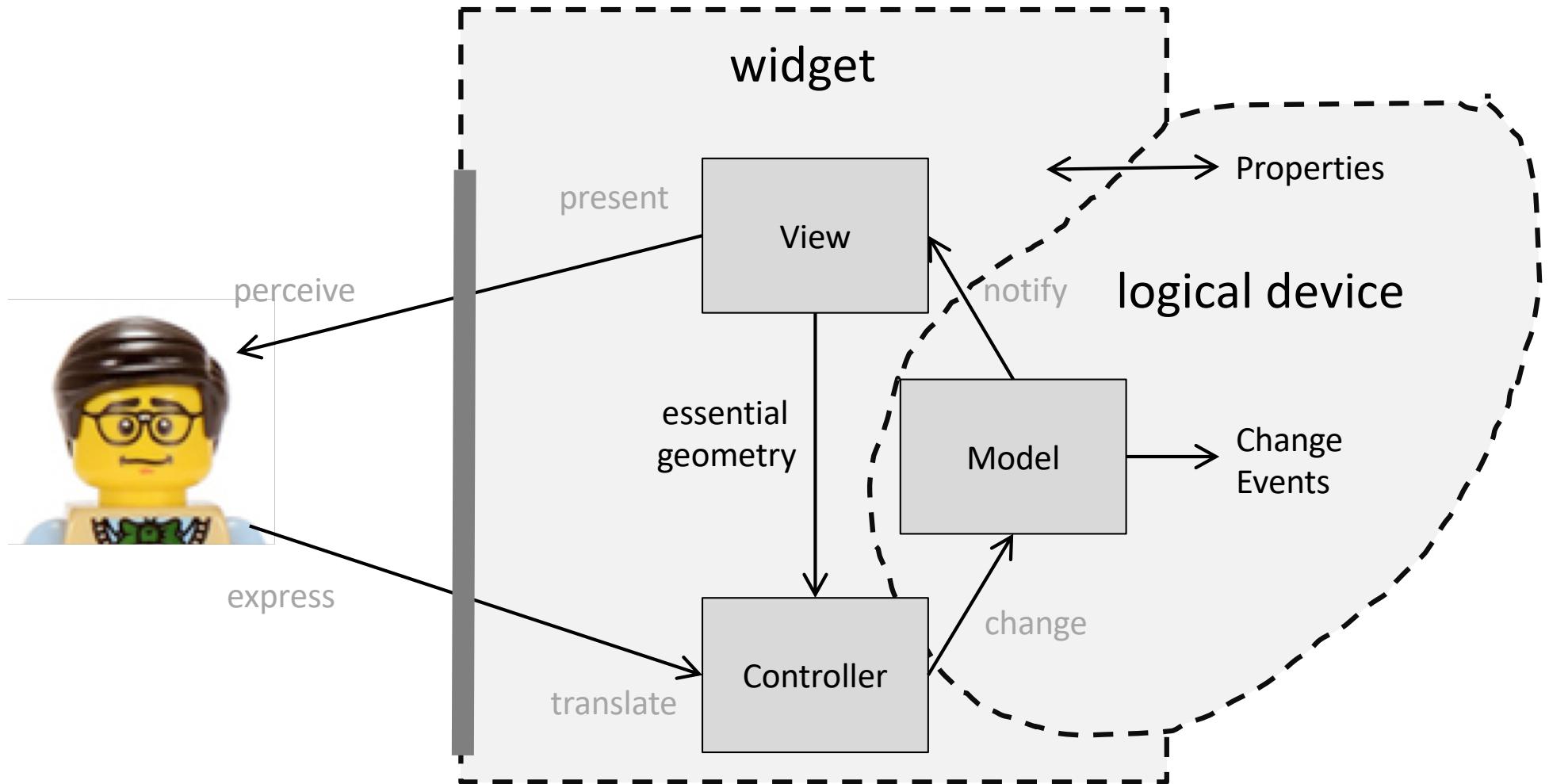
MVC Widget Architecture

Revisiting widgets

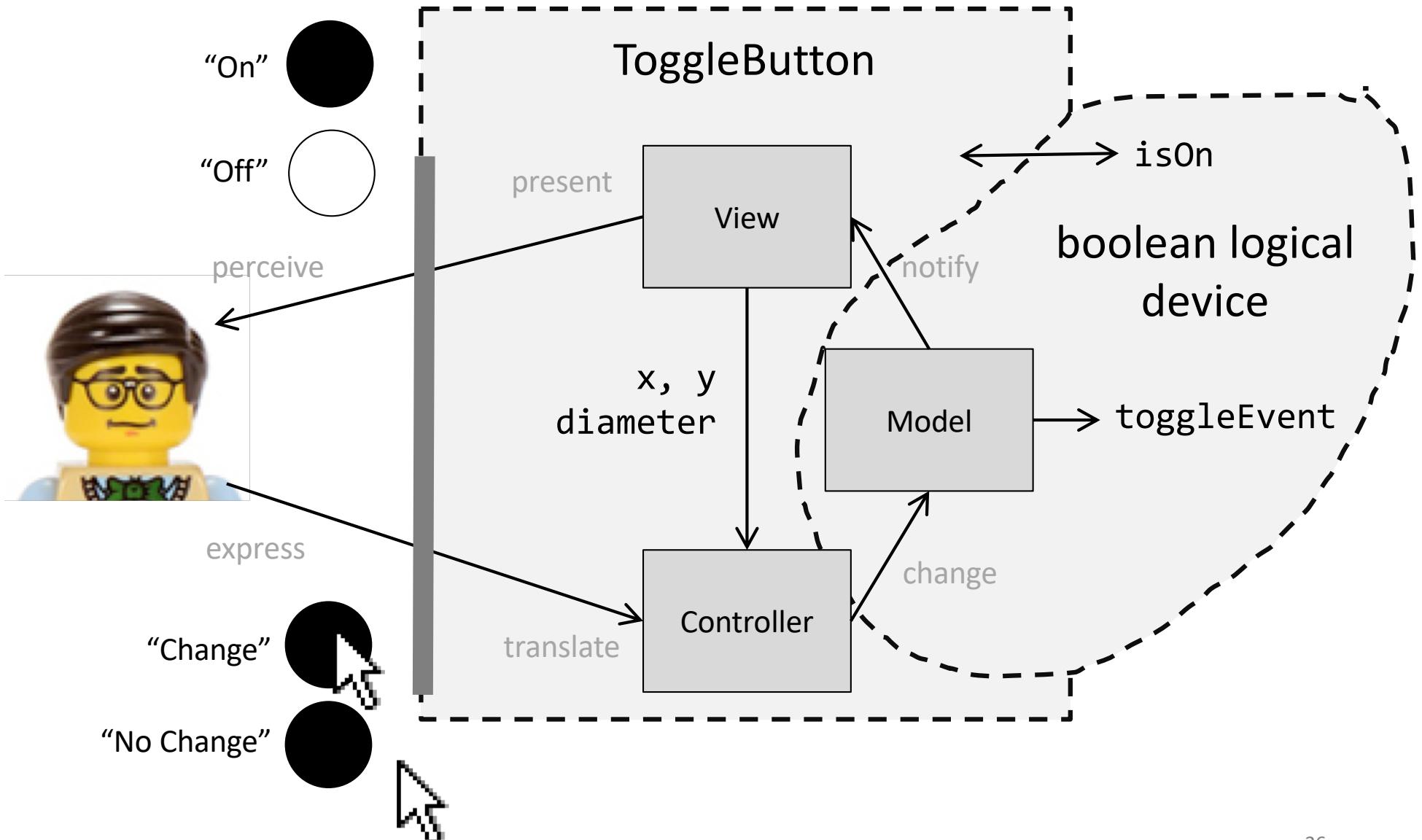
Widget Architecture as MVC



Widget Architecture as MVC



Custom Widgets: ToggleButton



ToggleButton.cpp

```
class ToggleButton {  
    ToggleButton(int _x, int _y, void (*_toggleEvent)(bool)) {  
        toggleEvent = _toggleEvent;  
        isOn = false;  
        ...  
    }  
  
    // the CONTROLLER  
    void mouseClicked(int mx, int my) {  
        float dist = sqrt(pow(mx - x, 2) + pow(my - y, 2));  
        if (dist < diameter) { toggle(); }  
    }  
  
    ...
```

ToggleButton.cpp (cont'd)

...

```
// the VIEW
void draw() {
    if (isOn) {
        setForeground(BLACK);
        XFillArc(...);
    } else {
        setForeground(WHITE);
        XFillArc(...);
    }
}
```

```
// VIEW "essential geometry"
int x;
int y;
int diameter;
```

...

ToggleButton.cpp (cont'd)

```
...  
  
// toggle event callback  
void (*toggleEvent)(bool);  
  
// the MODEL  
bool isOn;  
void toggle() {  
    isOn = !isOn;  
    toggleEvent(isOn); }  
};
```

ToggleButton.cpp (cont'd)

```
bool isPaused = false;
// isPaused callback (a simple event handler)
void togglePause(bool isOn) {
    isPaused = isOn;
}

...
ToggleButton toggleButton(150, 100, &togglePause);

...
case ButtonPress:
    toggleButton.mouseClick(event.xbutton.x, event.xbutton.y);
    break;

...
if (!isPaused) {
    // update ball position
}

toggleButton.draw();
```