Model View Controller

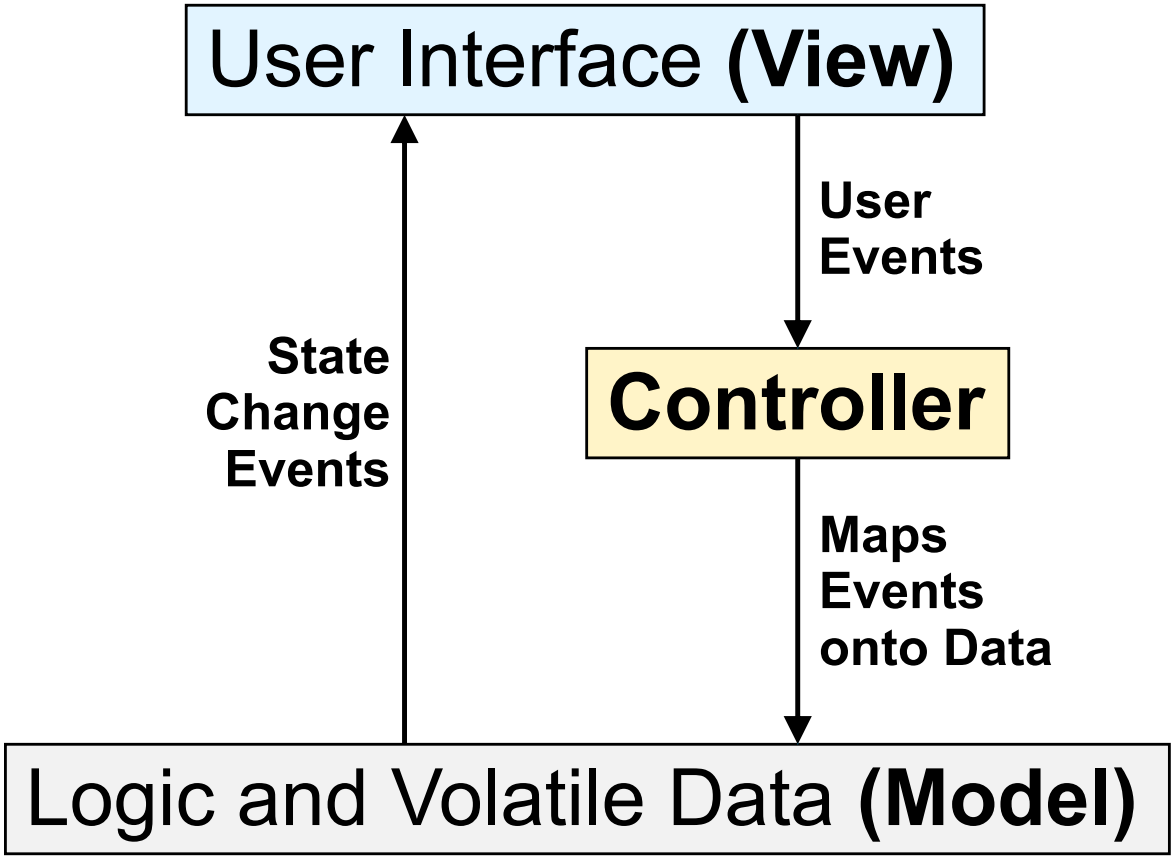
Michael Lones
EM G31 m.lones@hw.ac.uk
Material available through Vision

MVC Outline

This pattern separates a program's data, its display, and how it handles user interaction

- The **Model**
  - The underlying data of the program
  - and any associated logic (i.e. methods)
- The **View**
  - The visual display/presentation of the model
- The **Controller**
  - Handles user interaction with the model

Model, View & Controller



- This pattern says that you should implement the model, view and controller as **separate modules** that are largely independent of one another

Compound Design Pattern

MVC uses three other design patterns, which we've already covered in the lectures

- **Observer**
  - This is the main glue of MVC
- **Strategy**
  - This deals with interchangeable components
- **Composite**
  - Handles interface updates, at least when the GUI is a composite of components (as in Java)

Compound Design Pattern

MVC uses three other design patterns, which we've already covered in the lectures

- For this reason, it is known as a **compound design pattern**
- Also referred to as an **architectural pattern**, since it has a broader scope than most design patterns

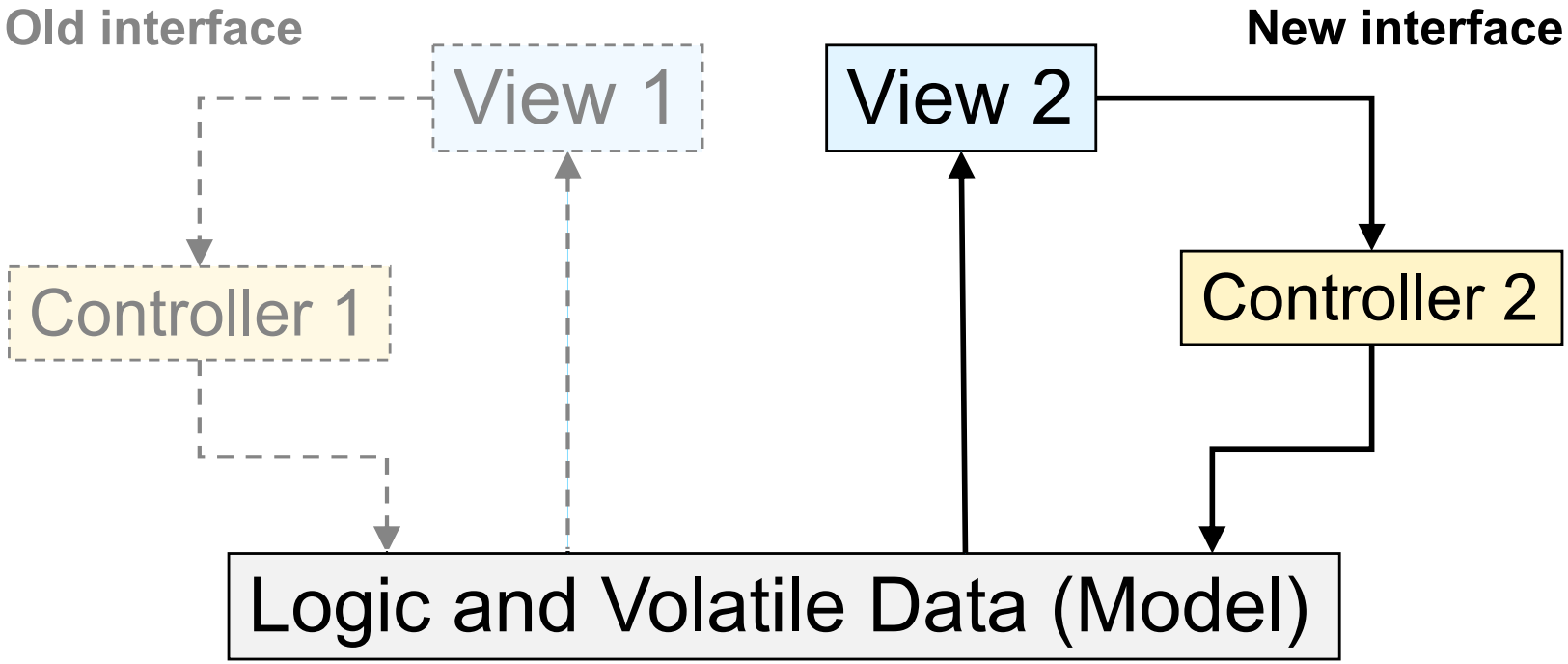
Benefits

MVC provides:

- **Clarity** in design, since the different parts of the program are separated. This aids maintainability.
- **Low coupling** between the models, views and controllers. Multiple developers can work at the same time on different parts of the system.
- **Extensibility**: controllers and views can be added and removed without affecting the model.
- **Modularity** so that components can be swapped, and multiple views on the same model data.

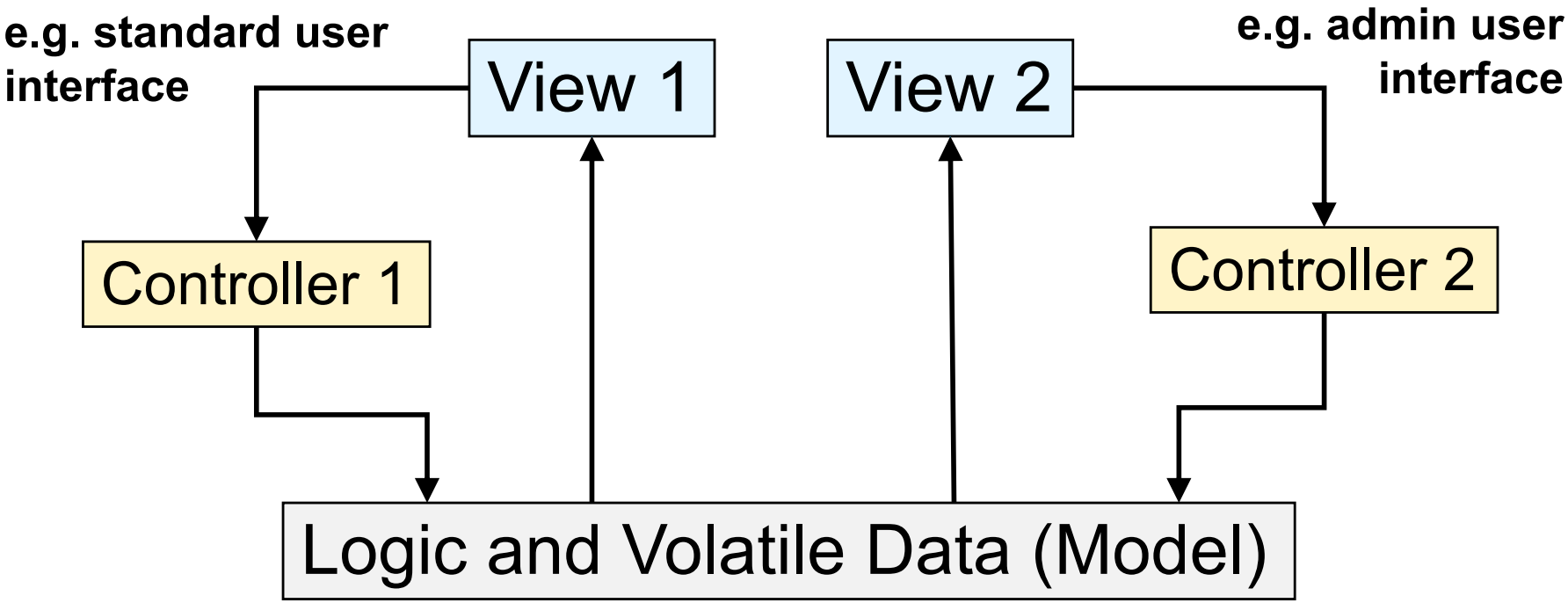
Model, View & Controller

- MVC makes it easy to update the interface of a program, since interface code is already separated



Model, Views & Controllers

- MVC also allows a system to have multiple interchangeable Views and Controllers

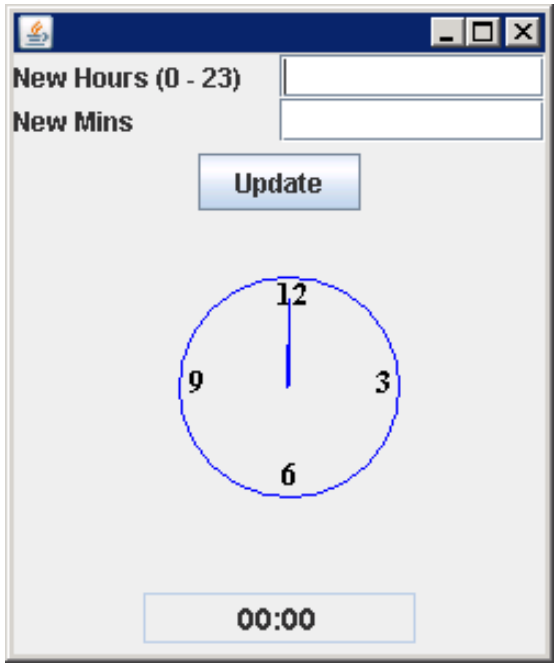


Clock Example

To demonstrate MVC, we are going back to the clock example from the Observer lecture

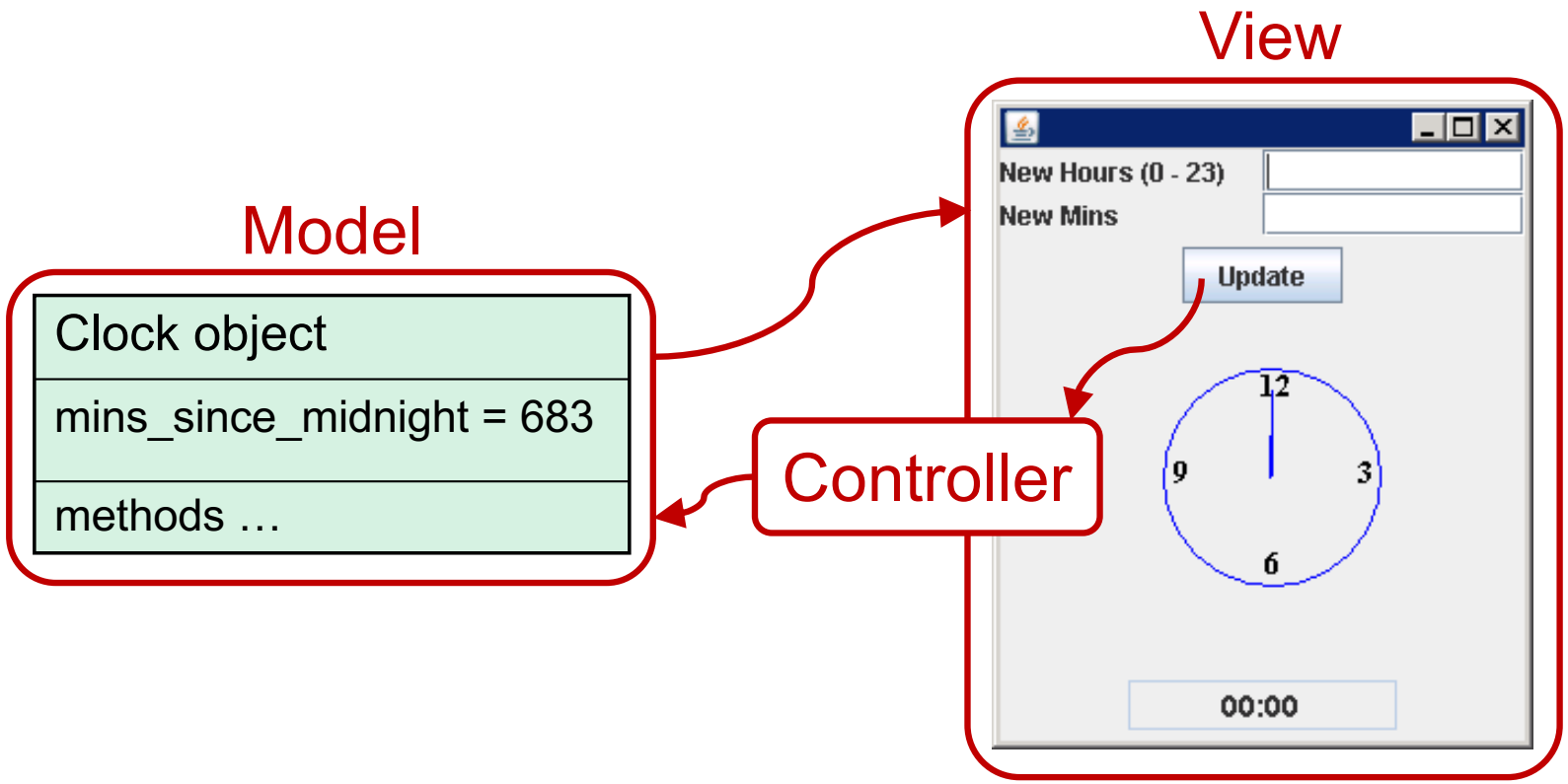
- Though note that the GUI elements have been combined, and we're not using the Counter class

Clock object
mins_since_midnight = 683
methods …



Clock Example

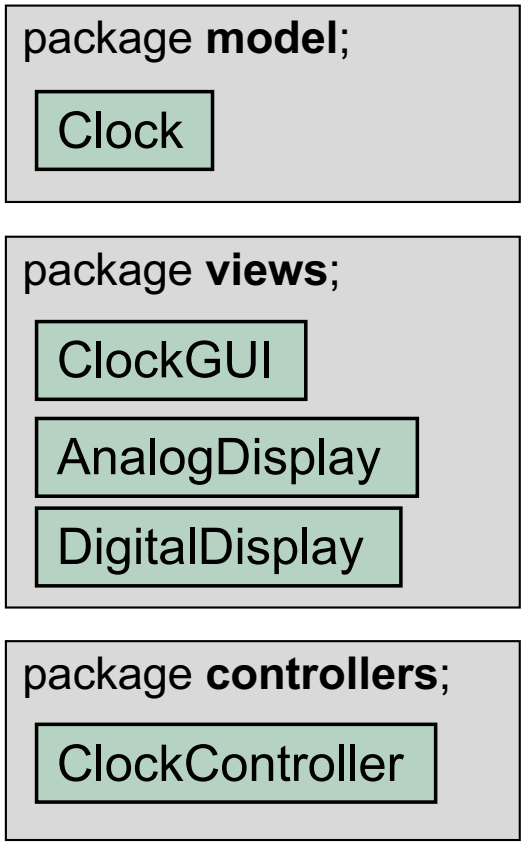
To demonstrate MVC, we are going back to the clock example from the Observer lecture



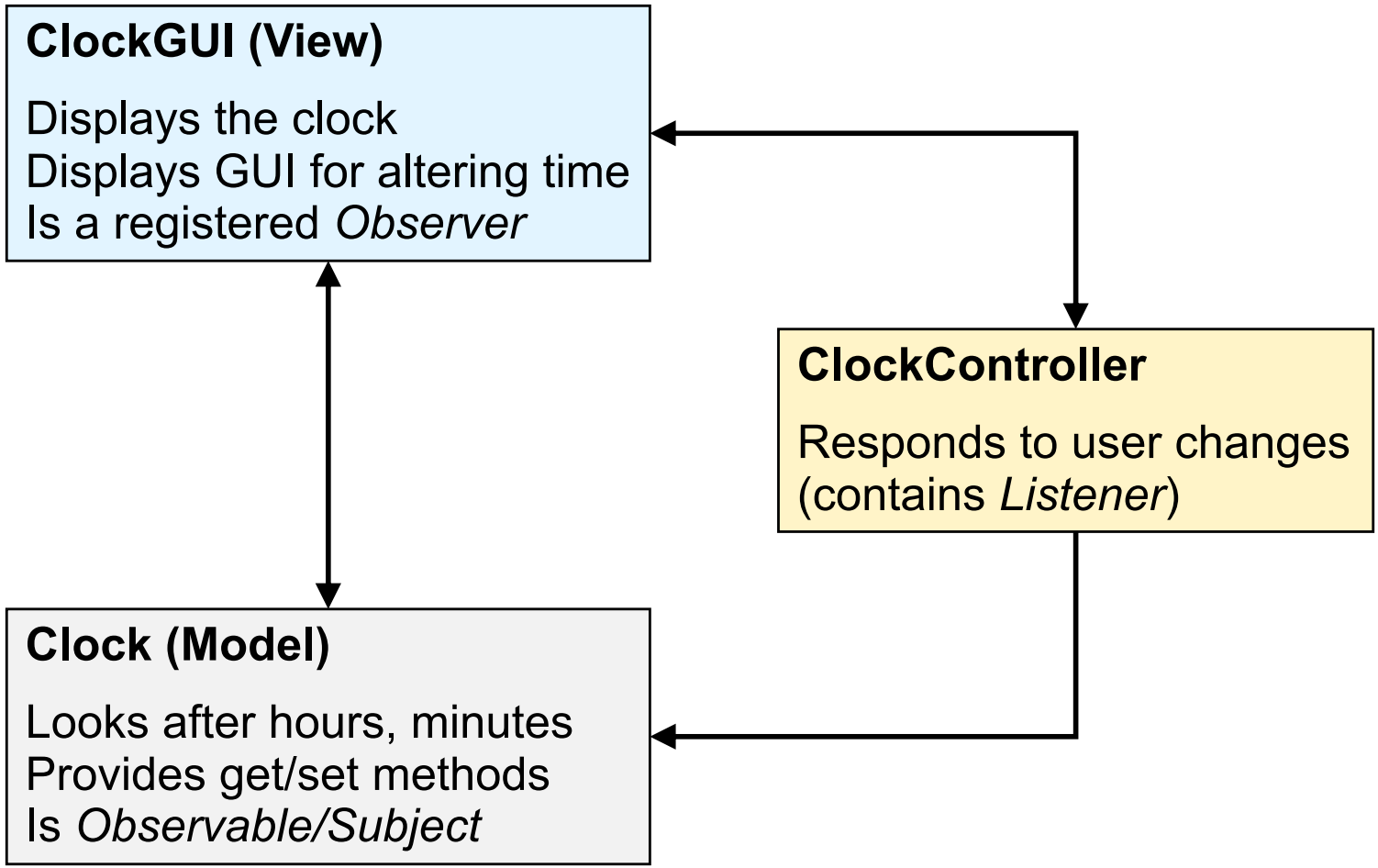
Code Separation

In the Clock MVC example supplied, the classes are arranged in packages

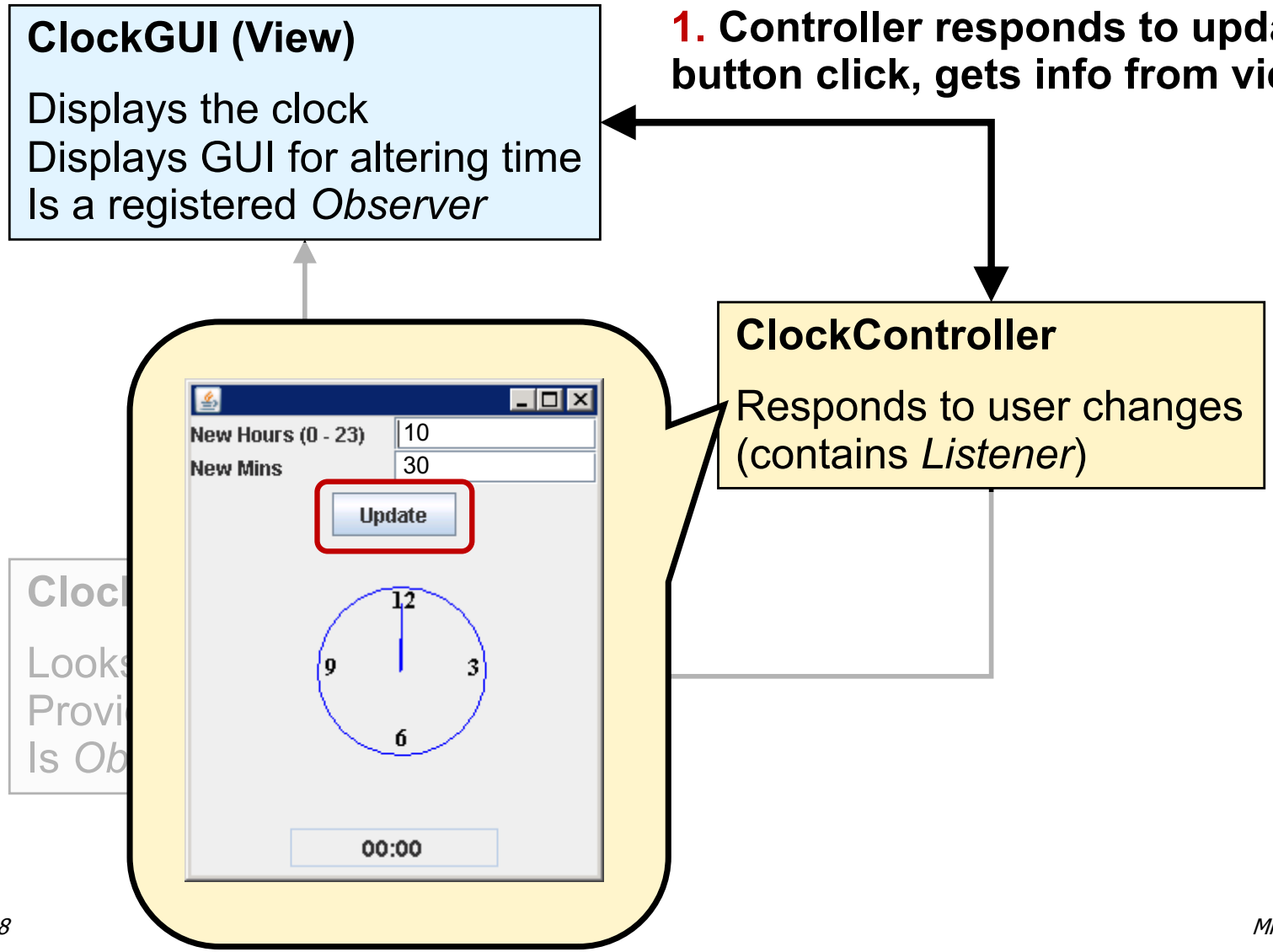
- Clock in the **`model`** package
- All the display classes in the **`views`** package
- The controller classes in **`controllers`**
- Also **`main`** and **`interfaces`** packages



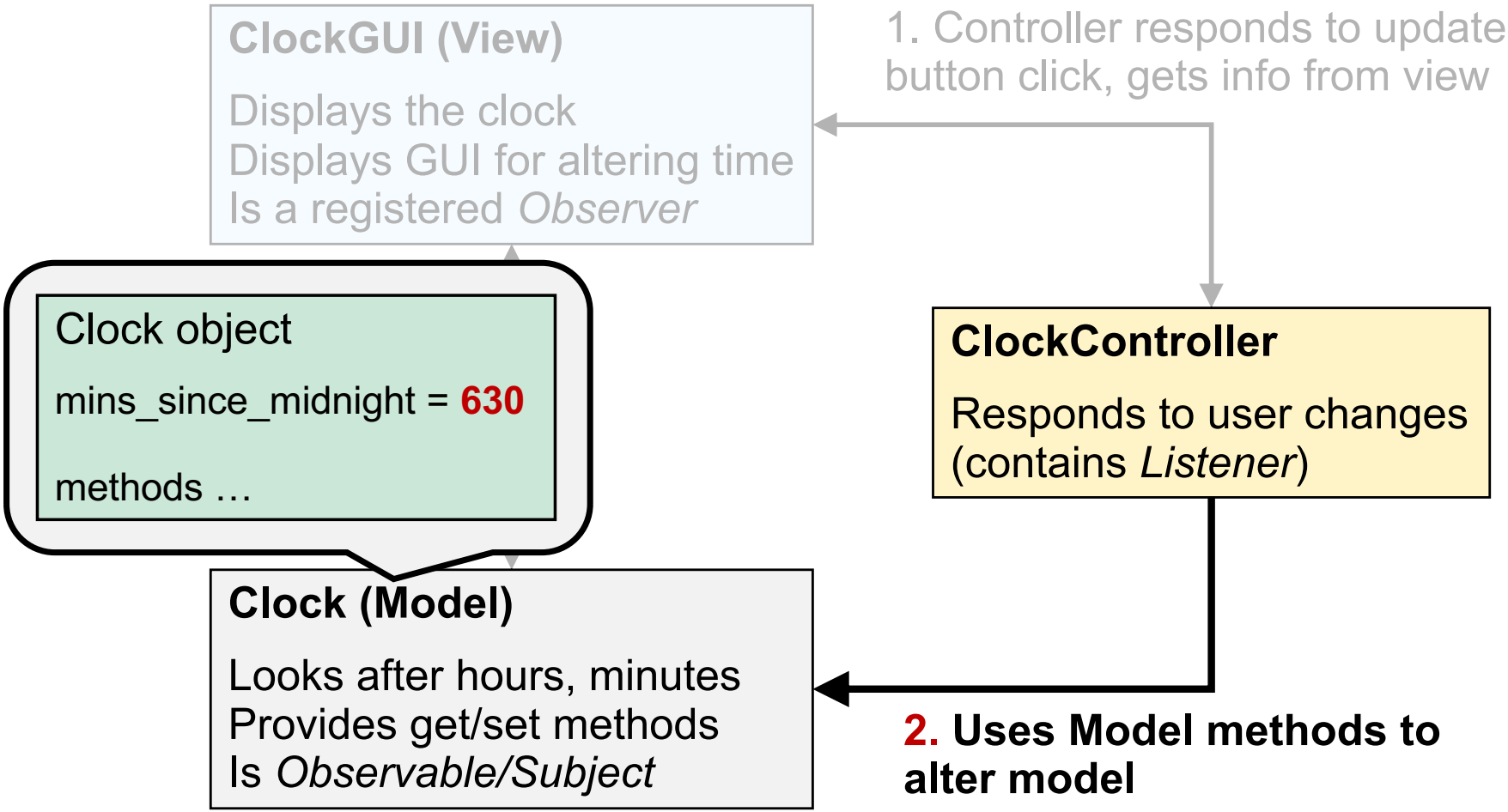
Responsibilities



Interactions



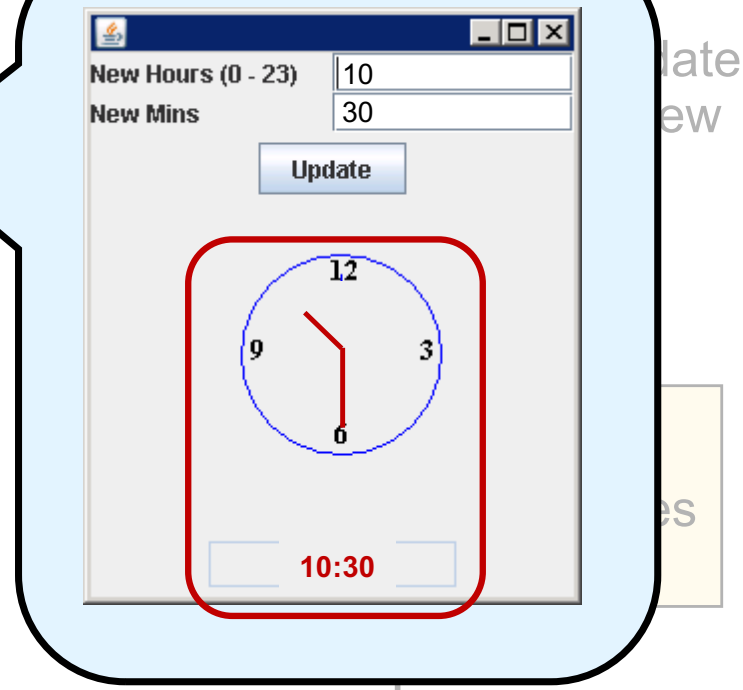
Interactions



Interactions

ClockGUI (View)

Displays the clock
Displays GUI for altering time
Is a registered *Observer*



3. Model notifies view to update (Observer pattern) and view gets data from model

Clock (Model)

Looks after hours, minutes
Provides get/set methods
Is *Observable/Subject*

2. Uses Model methods to alter model

Main Method

This is a classic MVC main method

ClockExample.java

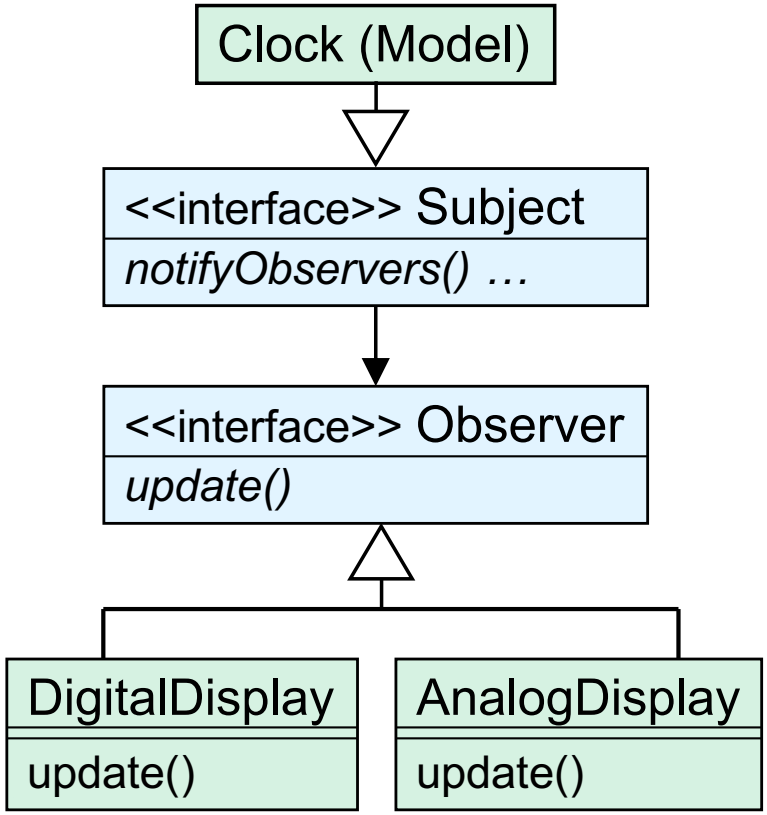
```
public static void main(String[] args) {
    Clock model = new Clock();
    ClockGUI view = new ClockGUI(model);
    ClockController controller = new
                    ClockController(view, model);
}
```

- Model does not know about View or Controllers
- The View only needs to know about the Model
- Only the Controller needs to know the details of all the other parts of the system

Use of the Observer Pattern

This allows observers to be updated about changes to a subject they are registered with

- The **Model** is a Subject
- The **Views** are Observers
- When the model changes, the views get updated
- In the Clock example, the observer pattern is used as before – the displays are Observers



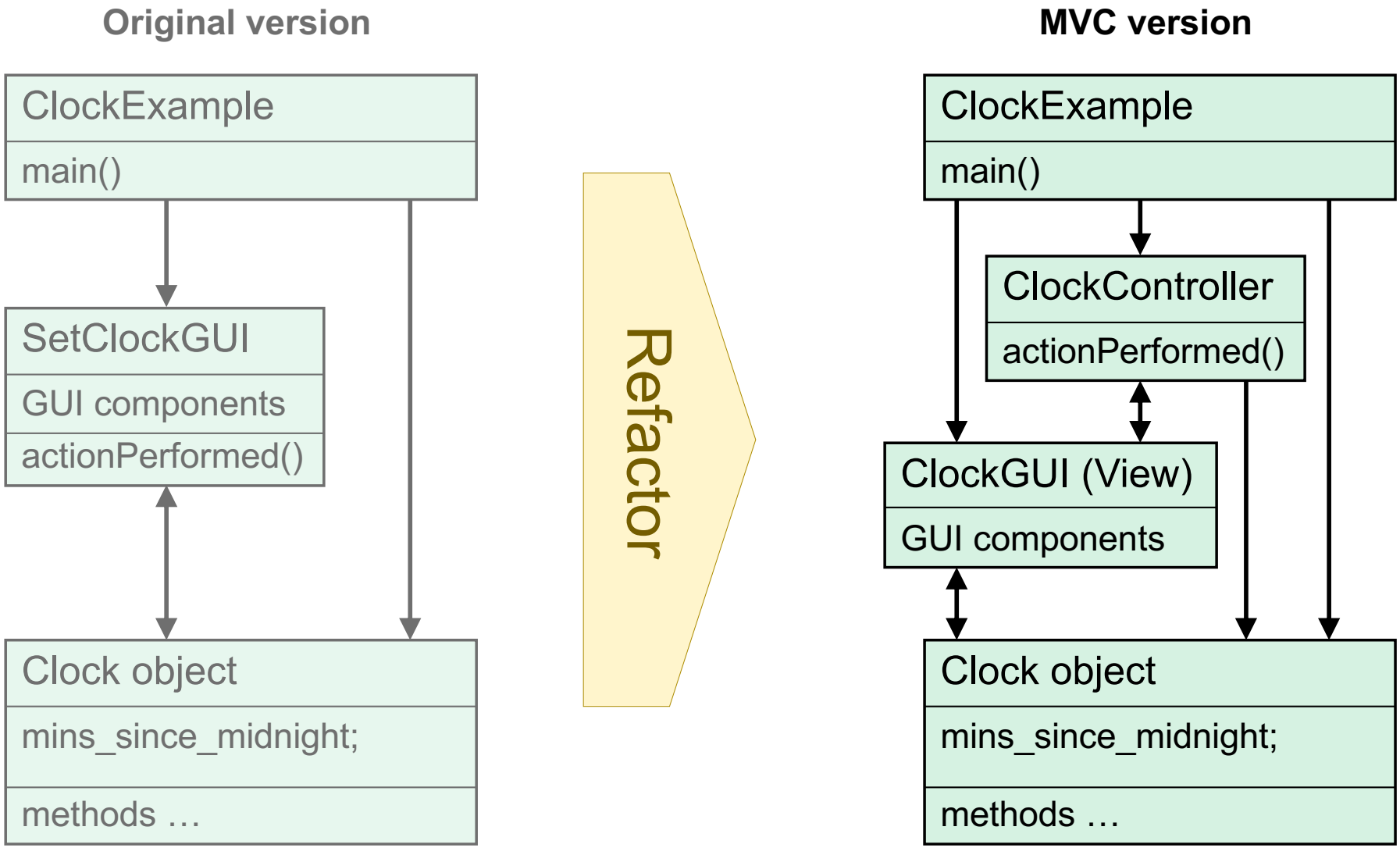
Refactoring the Clock Example

In the original clock example, View and Controller were combined in the same class:

```
class SetClockGUI extends JFrame implements ActionListener {
  // set up interface components, including update button
  JButton updateButton = new JButton("Update");

  public SetClockGUI(Clock clock) {
      // initialisation including
      updateButton.addActionListener(this);
  }

  public void actionPerformed (ActionEvent e) {
      // controller code that responds to update button
  }
}
```

SetClockGUI.java

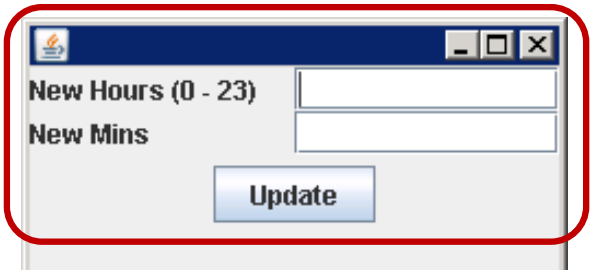
Refactoring the Clock Example



Refactoring the Clock Example

Using MVC, the View (ClockGUI) is only responsible for allowing the user to enter the hours and minutes and press update

- It has no responsibility for actions after that
- It provides a method to register the object that does, i.e. the controller
- And provides get methods for the controller to determine values entered by the user



ClockGUI
GUI components
getHours() : String **getMinutes**() : String **addSetListener**(Listener)

Refactoring the Clock Example

This is the new version (changes in red):

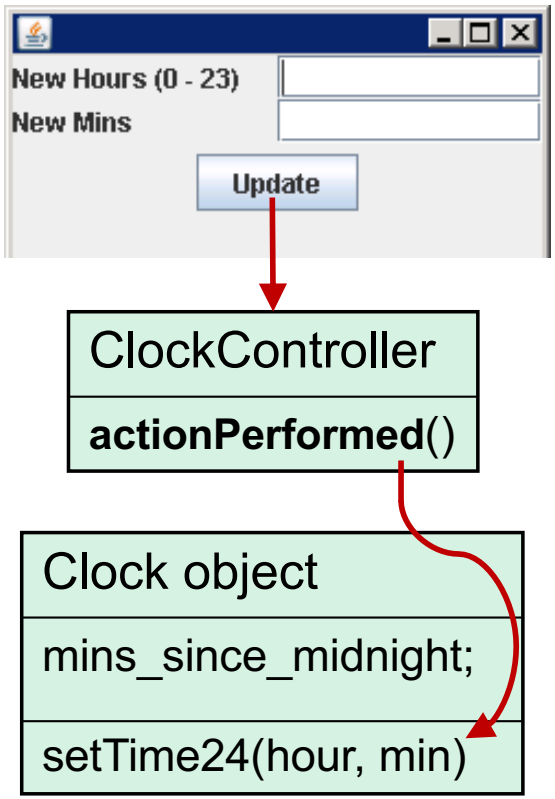
```
class SetClockGUI extends JFrame implements ActionListener {
  // set up interface components, including update button
  JButton updateButton = new JButton("Update");

  public ClockGUI(Clock clock) { //GUI initialisation code }

  // methods giving controller access to view data
  public String getHours() { return hours.getText();}
  public String getMins()  { return mins.getText(); }

  // allow listener to be added to update button
  public void addSetListener(ActionListener al) {
       updateButton.addActionListener(al);
  }
}
```

ClockGUI.java

Refactoring the Clock Example

Code for responding to user interaction goes in a new ClockController class

- Registers itself as an ActionListener for the View class's Update button
- Provides an actionPerformed method with code that responds to user's interaction
- Accesses the values in the text fields (using ClockGUI's getters) and updates the clock data



Refactoring the Clock Example

ClockController.java

```
public class ClockController {
   private ClockGUI view;  // view
   private Clock clock;    // model

   public ClockController(ClockGUI view, Clock clock) {
       this.clock = clock;   this.view = view;
       // specify the listener for the view
       view.addSetListener( new SetListener() );
   }

   // inner class SetListener responds when user sets time
   public class SetListener implements ActionListener {
       public void actionPerformed(ActionEvent e) {
           int hour = Integer.parseInt( view.getHours() );
           int min = Integer.parseInt( view.getMins() );
           clock.setTime24(hour, min);
  }
}
```

Refactoring the Clock Example

Why do we use an inner class here?

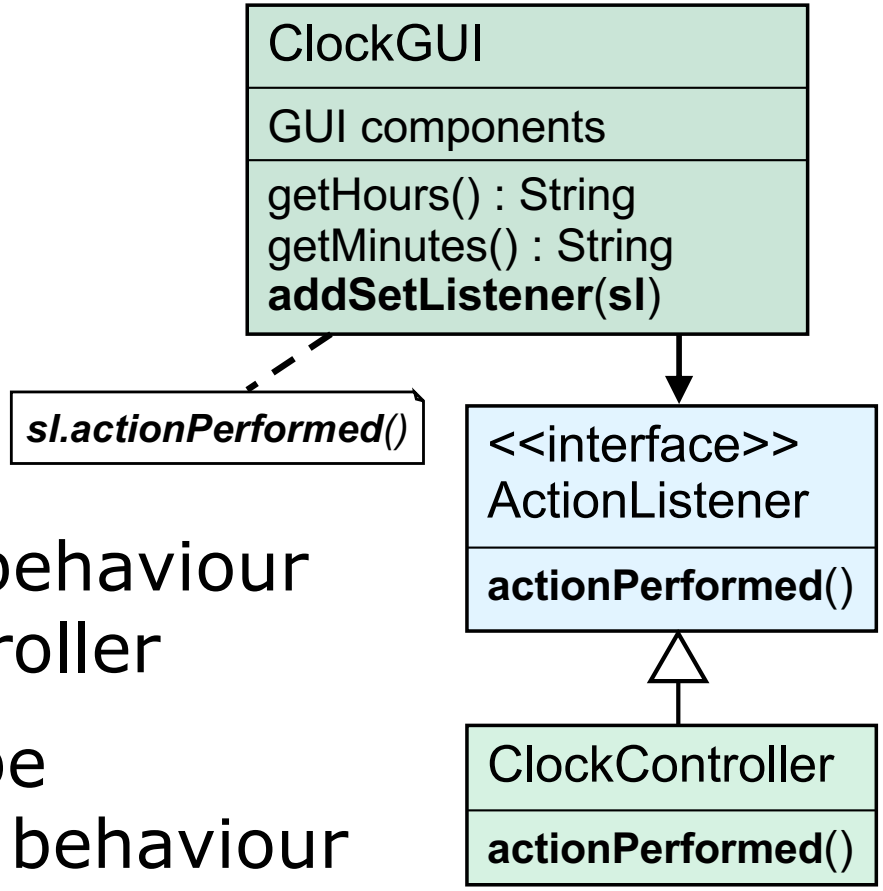
- We can have multiple inner classes, so (in a larger system) we could logically separate parts of the controller that deal with different parts of the interface.

```
        view.addSetListener( new SetListener() );
    }

    // inner class SetListener responds when user sets time
    public class SetListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            int hour = Integer.parseInt( view.getHours() );
            int min = Integer.parseInt( view.getMins() );
            clock.setTime24(hour, min);
    }
}
```

Use of the Strategy Pattern

The Strategy pattern allows you to select one of several algorithms dynamically

- The **View** and **Controller** implement this pattern, i.e. the View is configured with a Controller strategy
- The View only needs to be concerned with visual aspects; decisions about behaviour are delegated to the Controller
- Another Controller could be swapped in to change the behaviour



Use of the Composite Pattern

All components are treated the same, whether they are composites or leaf nodes

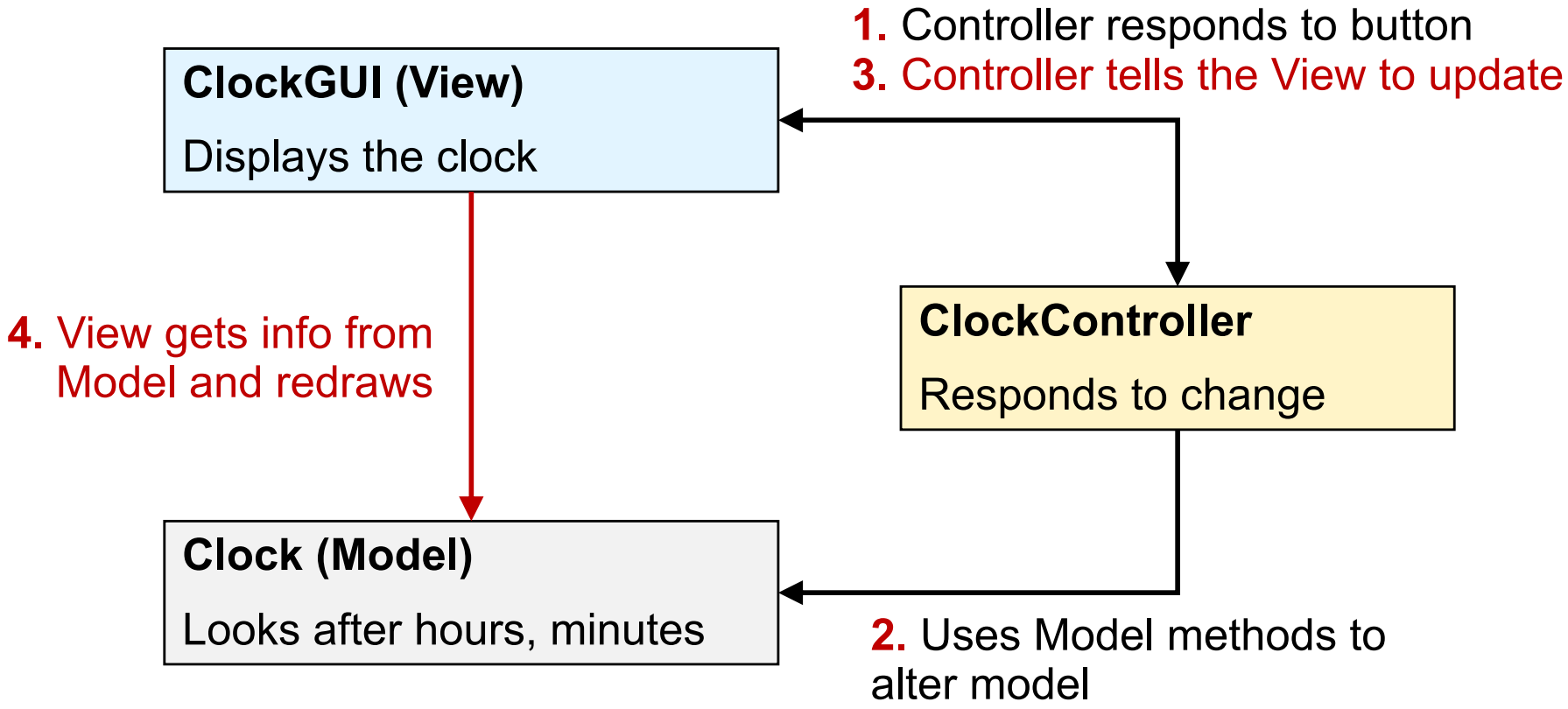
- The **View** is usually implemented using Composite
- In Swing, GUIs are already composites, so we don't need to explicitly include this pattern
- When a top-level component is updated, it uses composite to update all lower-level components (using JComponent's paintComponent method)
- In essence, we just need to register one observer per view, and the rest is updated automatically

MVC Variants

ASE

You might come across variants of MVC

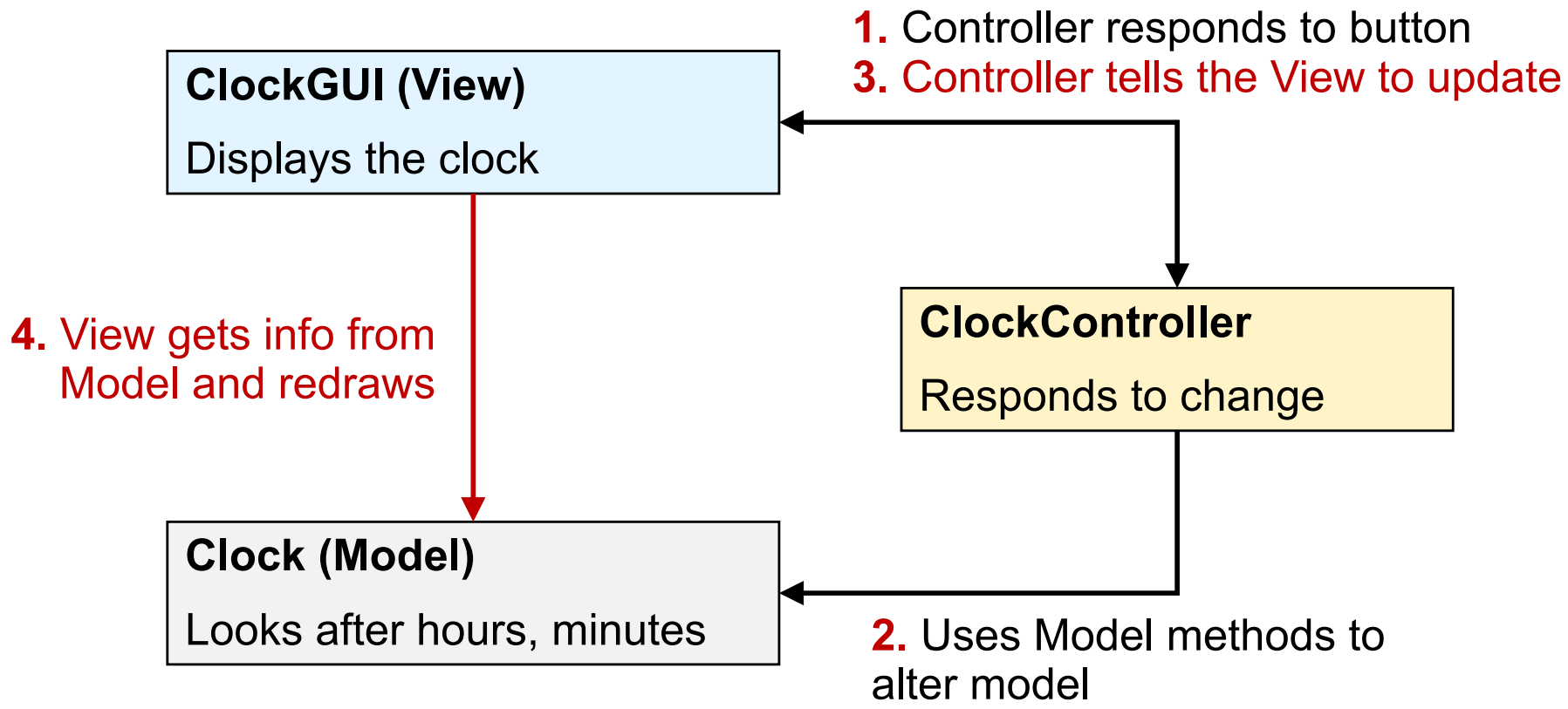
- E.g. here is a variant that doesn't use Observer…



MVC Variants

What happens if the Model changes internally?

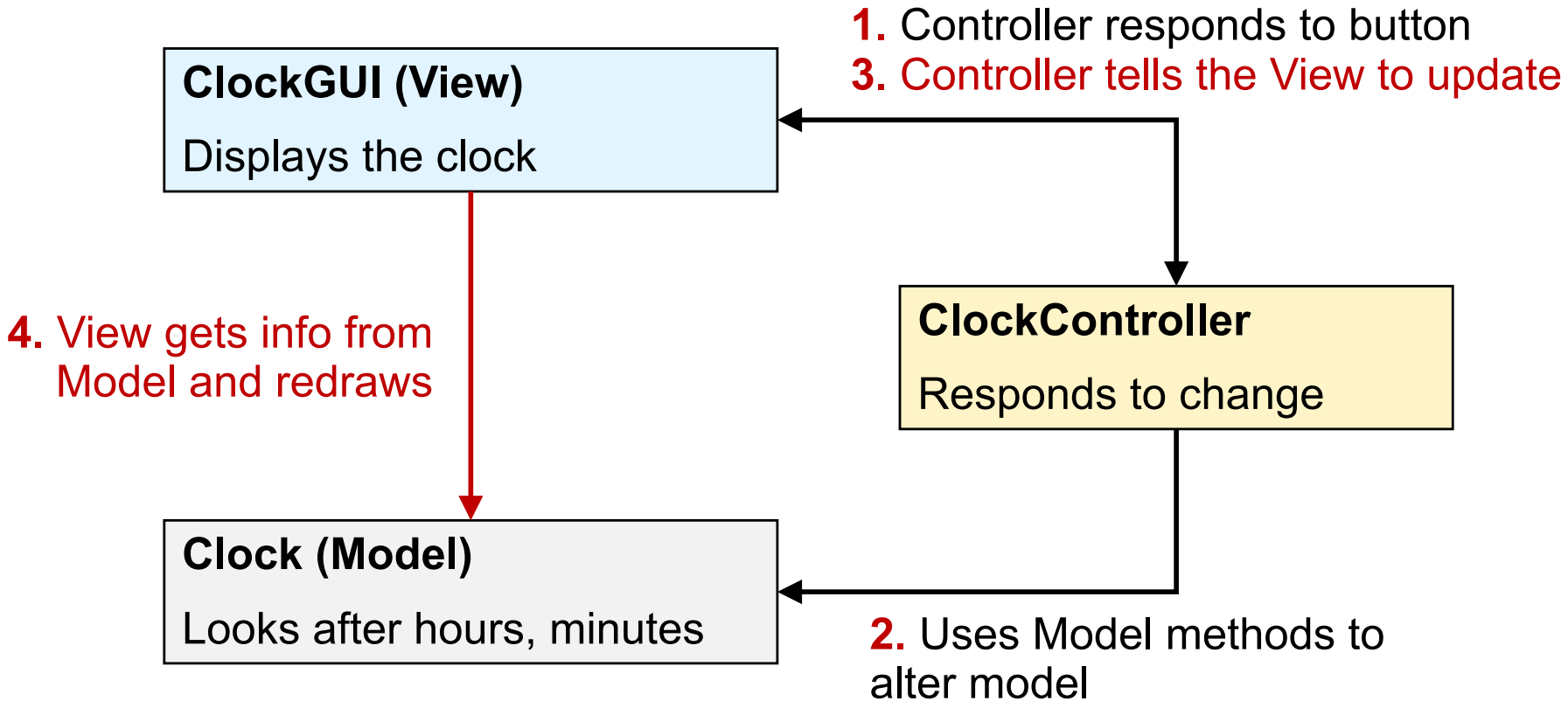
- The Views will not automatically be notified ☹



MVC Variants

What if there are multiple Controllers?

- Only the associated view gets updated, not others ☹



MVC in Swing

Java uses a "Modified MVC Architecture"

- Basically, for most JComponents, the View and Model are coupled together
- E.g. a JLabel contains an underlying Model (the label text and/or image) as well as being its View
- For interactive components, such as JButtons, programmers can add their own Controller (using the appropriate Listener interface)
- For more complex components, such as JTables and JLists, you can provide a separate Model (e.g. by using the TableModel interface)

Summary

MVC is a widely-used architectural pattern

- It's a compound pattern, often including the Observer, Strategy and Composite patterns
- In general, it makes it easier to write code that is modular and maintainable
- MVC is often found in GUI libraries and web frameworks, e.g. Java's Swing, ASP.NET

MVC…

has been called the King of Patterns

- There's even a song by James Dempsey
- http://www.youtube.com/watch?v=YYvOGPMLVDo

MVC Song text

- http://www.oreillynet.com/onjava/blog/2003/07/new_way_to_learn_mvc_view_a_si.html

"Model View, Model View, Model View Controller
MVC's the paradigm for factoring your code,
into functional segments so your brain does not explode.
To achieve reusability you gotta keep those boundaries clean,
Model on the one side, View on the other, the Controller's in between.

Model View - It's got three layers like Oreos do.
Model View creamy Controller

Model objects represent your applications raison d'tre.
Custom classes that contain data logic and et cetra.
You create custom classes in your app's problem domain,
then you can choose to reuse them with all the views,
but the model objects stay the same"