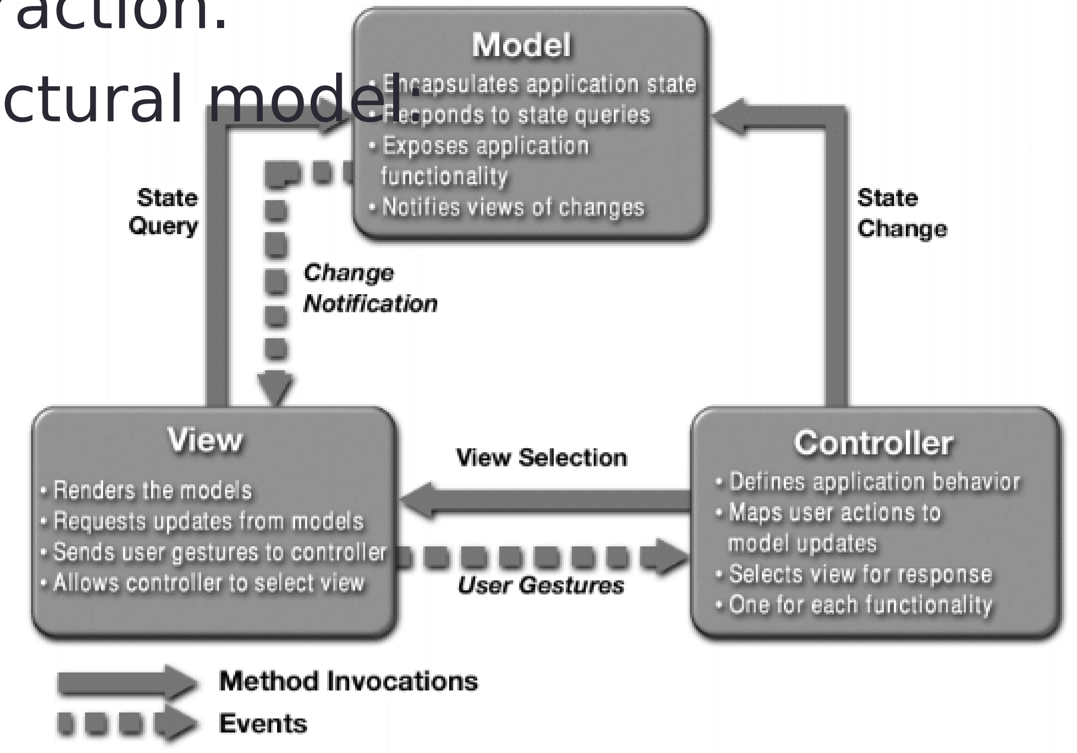
ADVANCED PROGRAMMING PRACTICES

Model View Controller Architecture

Observer Pattern

Model View Controller Architecture

- MVC was first introduced by Trygve Reenskaug at the Xerox Palo Alto Research Center in 1979.
- Part of the basic of the **Smalltalk** programming environment.
- Widely used for many object-oriented designs involving user interaction.
- A three-tier architectural model


Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

MVC: model

- Model:
  - Manages the behavior and data of the application domain.
  - Responds to requests for information about its state (usually from the view).
  - Responds to instructions to change state (usually from the controller).
  - In event-driven systems, the model notifies observers (usually views) when the information changes so that they can react. (see observer pattern)
  - In enterprise software, a model often serves as a software approximation of a real-world process.
  - In a game, the model is represented by the classes defining the game entities, which are embedding their own state and actions.

MVC: view

- View:
  - <u>Renders the model</u> into a form suitable for <u>visualization</u> or <u>interaction</u>, typically a user interface element.
  - Multiple views can exist for a single model element for different purposes.
  - The view renders the contents of a portion of the model's data.
  - If the model data changes, the view must update its presentation as needed. This can be achieved by using:
    - a **push model**, in which the view registers itself with the model for change notifications. (see the observer pattern)
    - a **pull model**, in which the view is responsible for calling the model when it needs to retrieve the most current data.

MVC: controller

- Controller:
  - Receives user input and initiates a <u>response</u> by <u>making calls on appropriate model objects</u>.
  - Accepts input (e.g. events or data) from the user and instructs the model to perform actions based on that input.
  - The controller translates the user's interactions with the view it is associated with, into actions that the model will perform that may use some additional/changed data gathered in a user-interactive view.
  - A controller may also spawn new views upon user demand.

MVC: interactions between model, view and controller

- Creation of a Model-View-Controller trio:
  1. The model objects are created.
     - Each model object represents a portion of the business model state held by the application.
  2. The views register as observers on the model objects.
     - Any changes to the underlying data of the model objects immediately result in a broadcast change notification, which all associated views receive (in the push model).
     - Note that the model is not aware of the view or the controller -- it simply broadcasts change notifications to all registered observers.
  3. The controller is bound to a view.
     - It can then react to any user interaction provided by this view.
     - Any user actions that are performed on the view will invoke a method in the controller class.
  4. The controller is given a reference to the underlying model.
     - It can then trigger the model's behavior functions and/or state change when one of its methods is called.

MVC: interactions between model, view and controller

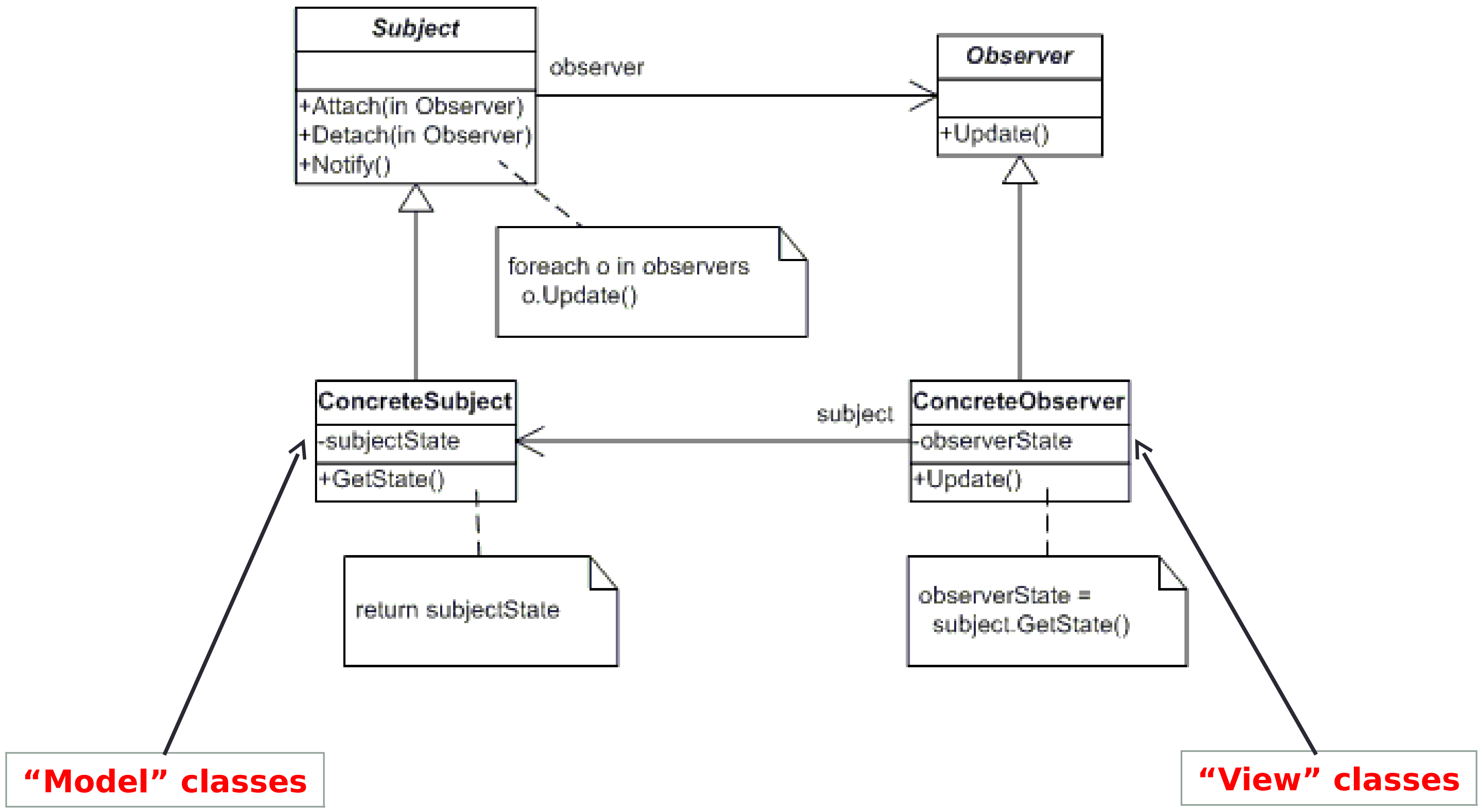
- Once a user interacts with the view, the following actions occur:
    1. The view recognizes that a GUI action -- for example, pushing a button or dragging a scroll bar -- has occurred, e.g using a listener method that is registered to be called when such an action occurs. The mechanism varies depending on the technology or library used.
    2. In the listener method, the view calls the appropriate method in the controller.
    3. The controller translates this signal into an appropriate action in the model, which will in turn possibly be updated in a way appropriate to the user's action.
    4. If the state of some of the model’s elements have been altered, they then notify registered observers of the change. In some architectures, the controller may also be responsible for updating the view. Again, technical details may vary according to technology or library used.

Observer pattern

Observer pattern: motivation, intent

- Motivation
  - The cases when certain objects need to be informed about the changes occurring in other objects are frequent. To have a good design means to decouple as much as possible and to reduce the dependencies. The Observer Design Pattern can be used whenever a subject has to be observed by one or more observers.
- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- This pattern is a cornerstone of the Model-View-Controller architectural design, where the Model implements the business logic of the program, and the Views are implemented as Observers that are as much uncoupled as possible from the Model components.

Observer pattern: design



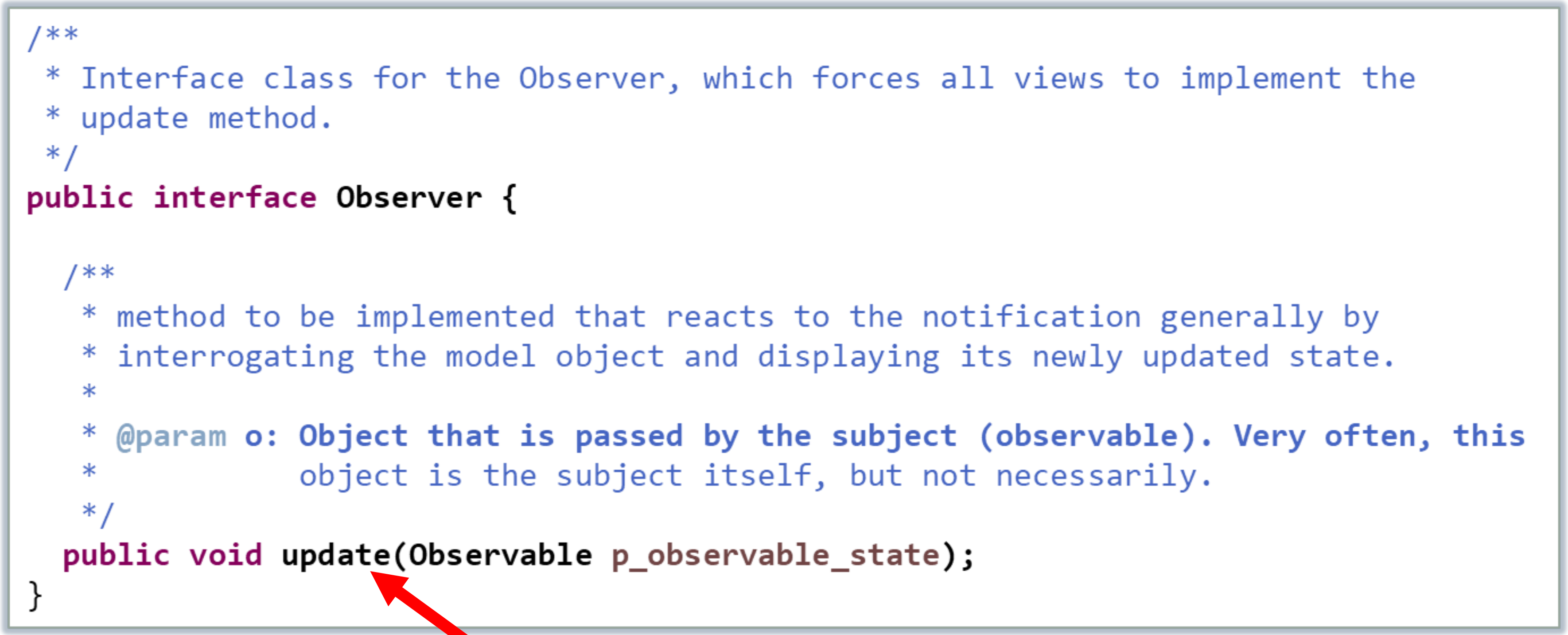
Observer pattern: design

- The participants classes in the Observer pattern are:
  - **Subject** - interface or abstract class defining the operations for attaching and de-attaching observers to the client. It is often referred to as “Observable”.
  - **ConcreteSubject** - concrete Subject class. It maintains the state of the observed object and when a change in its state occurs it notifies the attached Observers. If used as part of MVC, the ConcreteSubject classes are the Model classes that have Views attached to them.
  - **Observer** - interface or abstract class defining the operations to be used to notify the registered Observer objects.
  - **ConcreteObserver** - concrete Observer subclasses that are attached to a particular Subject class. There may be different concrete observers attached to a single Subject that will provide a different view of that Subject.

Observer pattern: behavior

- Behavior
  - The client class instantiates the ConcreteObservable object.
  - Then it instantiates and attaches the concrete observers to it using the methods defined in the Observable interface.
  - Each time the (observable) state of the subject is changing, it notifies all the attached Observers using the methods defined in the Observer interface.
  - When a new Observer is added to the application, all we need to do is to instantiate it in the client class and to add attach it to the Observable object.
  - The classes already created will remain mostly unchanged.

Observer pattern: implementation

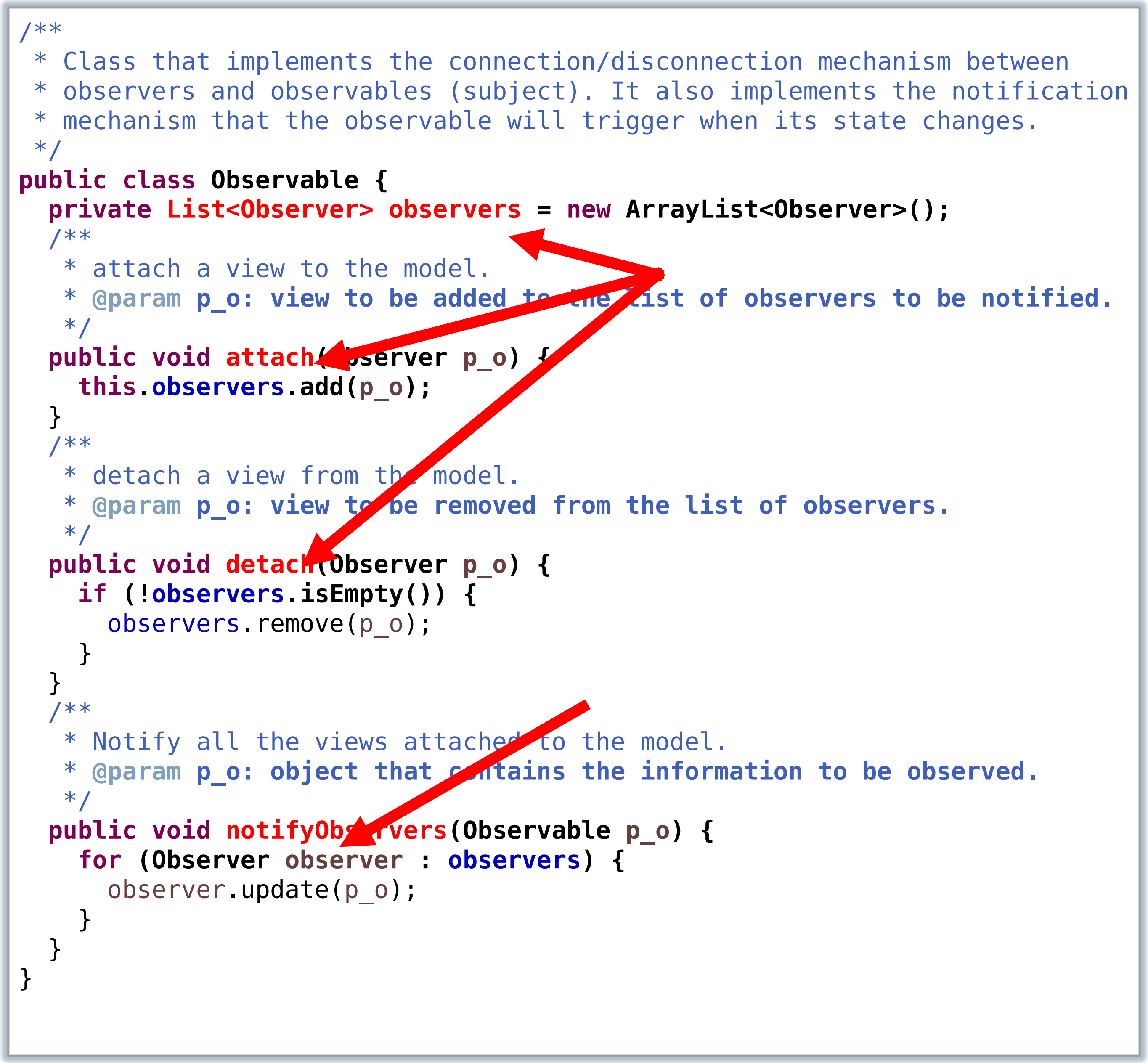


```
/**
 * Interface class for the Observer, which forces all views to implement the
 * update method.
 */
public interface Observer {

  /**
   * method to be implemented that reacts to the notification generally by
   * interrogating the model object and displaying its newly updated state.
   *
   * @param o: Object that is passed by the subject (observable). Very often, this
   *           object is the subject itself, but not necessarily.
   */
  public void update(Observable p_observable_state);
}
```

- Observer is an <u>interface</u> (it may also be an abstract class).
- Declares a method `update()` that is used polymorphically.
- All classes implementing this interface it must then override this method.

Observer pattern: implementation



```
/**
 * Class that implements the connection/disconnection mechanism between
 * observers and observables (subject). It also implements the notification
 * mechanism that the observable will trigger when its state changes.
 */
public class Observable {
  private List<Observer> observers = new ArrayList<Observer>();
  /**
   * attach a view to the model.
   * @param p_o: view to be added to the list of observers to be notified.
   */
  public void attach(Observer p_o) {
    this.observers.add(p_o);
  }
  /**
   * detach a view from the model.
   * @param p_o: view to be removed from the list of observers.
   */
  public void detach(Observer p_o) {
    if (!observers.isEmpty()) {
      observers.remove(p_o);
    }
  }
  /**
   * Notify all the views attached to the model.
   * @param p_o: object that contains the information to be observed.
   */
  public void notifyObservers(Observable p_o) {
    for (Observer observer : observers) {
      observer.update(p_o);
    }
  }
}
```

- The `Observable` base class is providing the implementation of the notification mechanism, including the attach/detach mechanism.

Observer pattern: implementation

```
class ClockTimerModel extends Observable {

  private int hour;
  private int minute;
  private int second;
  private int timedInterval;

  public int getHour() {return hour;};
  public int getMinute() {return minute;};
  public int getSecond() {return second;};
  public void tick() {
    second++;
    if (second >= 60) {
      minute++; second = 0;
      if (minute >= 60) {
        hour++; minute = 0;
        if (hour >= 24) {
          hour = 0;
        };
      };
    };
    notifyObservers(this);
  };

  public void start() {
    for (int i = 1; i <= timedInterval; i++)
      tick();
  };

  public void setTime(int h, int m, int s) {
    hour = h; minute = m; second = s;
    notifyObservers(this);
  }

  public void setTimedInterval(int t) {
    timedInterval = t;}
  };
```

- The Model classes implement the business model of the implementation.
- In order to be made `ConcreteObservable` classes, they have to:
  - Inherit the attach/detach and notification mechanisms from the `Observable` class.
  - Notify their `Observers` when an observable part their state changes.

Observer pattern: implementation

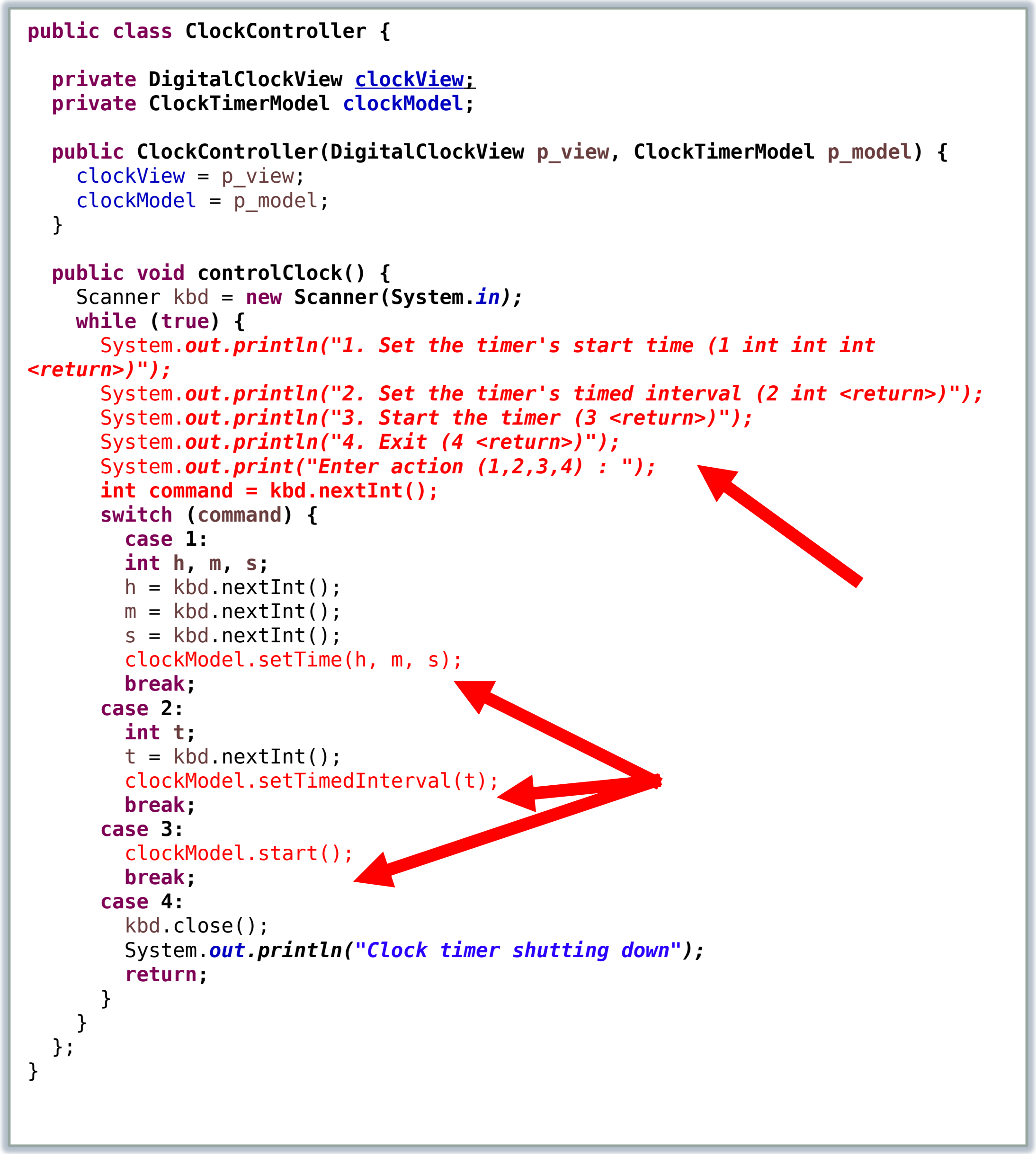
```
/**
 * This is the View class of the MVC trio for the clock
 * timer example.
 */
class DigitalClockView implements Observer {

  /**
   * Constructor that attaches the model to the view.
   *
   * @param clockModel
   */
  public DigitalClockView(ClockTimerModel p_clockModel) {
    p_clockModel.attach(this);
  }

  /**
   * Display the new hour, minute, second of the clock after
   * the view has been notified of a state change in the model.
   *
   * @param obs: object that contains the displayed information
   * @return none
   */
  public void update(Observable p_o) {
    int hour = ((ClockTimerModel) p_o).getHour();
    int minute = ((ClockTimerModel) p_o).getMinute();
    int second = ((ClockTimerModel) p_o).getSecond();
    System.out.println(hour + ":" + minute + ":" + second);
  };
};
```

- The View classes implement the displaying of information relevant to the user from Model objects.
- In order to be made `ConcreteObserver` classes, they have to:
  - Implement the `Observer` interface.
  - Implement the `update()` method declared in the `Observer` interface.

Controller: implementation



```
public class ClockController {

  private DigitalClockView clockView;
  private ClockTimerModel clockModel;

  public ClockController(DigitalClockView p_view, ClockTimerModel p_model) {
    clockView = p_view;
    clockModel = p_model;
  }

  public void controlClock() {
    Scanner kbd = new Scanner(System.in);
    while (true) {
      System.out.println("1. Set the timer's start time (1 int int int
<return>)");
      System.out.println("2. Set the timer's timed interval (2 int <return>)");
      System.out.println("3. Start the timer (3 <return>)");
      System.out.println("4. Exit (4 <return>)");
      System.out.print("Enter action (1,2,3,4) : ");
      int command = kbd.nextInt();
      switch (command) {
        case 1:
        int h, m, s;
        h = kbd.nextInt();
        m = kbd.nextInt();
        s = kbd.nextInt();
        clockModel.setTime(h, m, s);
        break;
      case 2:
        int t;
        t = kbd.nextInt();
        clockModel.setTimedInterval(t);
        break;
      case 3:
        clockModel.start();
        break;
      case 4:
        kbd.close();
        System.out.println("Clock timer shutting down");
        return;
      }
    }
  };
}
```

- The Controller classes implement the control flow involved between the operation of the Model object and the View object
- In this example's case:
  - Interact with the user to get their input used to set the clock's operation.
  - Trigger some methods of the clock to trigger the clock's operation.
  - In this example, the controller does not interact with the View. In most cases, the Controller does interact with the

MVC: putting it all together

```java
public class MVCDemo extends Object {
  private DigitalClockView clockView;
  private ClockTimerModel clockModel;
  private ClockController clockController;

  public MVCDemo() {
    clockModel = new ClockTimerModel();
    clockView = new DigitalClockView(clockModel);
    clockController = new ClockController(clockView, clockModel);
  };

  public static void main(String[] av) {
    MVCDemo od = new MVCDemo();
    od.demo();
  };

  public void demo() {
    clockController.controlClock();
    clockModel.detach(clockView);
  };
};
```

- In this example, the Model/View/Controller objects are created and connected for the entire duration of the program's lifetime.
- Depending on the application, some Views/Controllers may be created/removed by user actions.

References

- OODesign.com. Observer Pattern.
- Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.