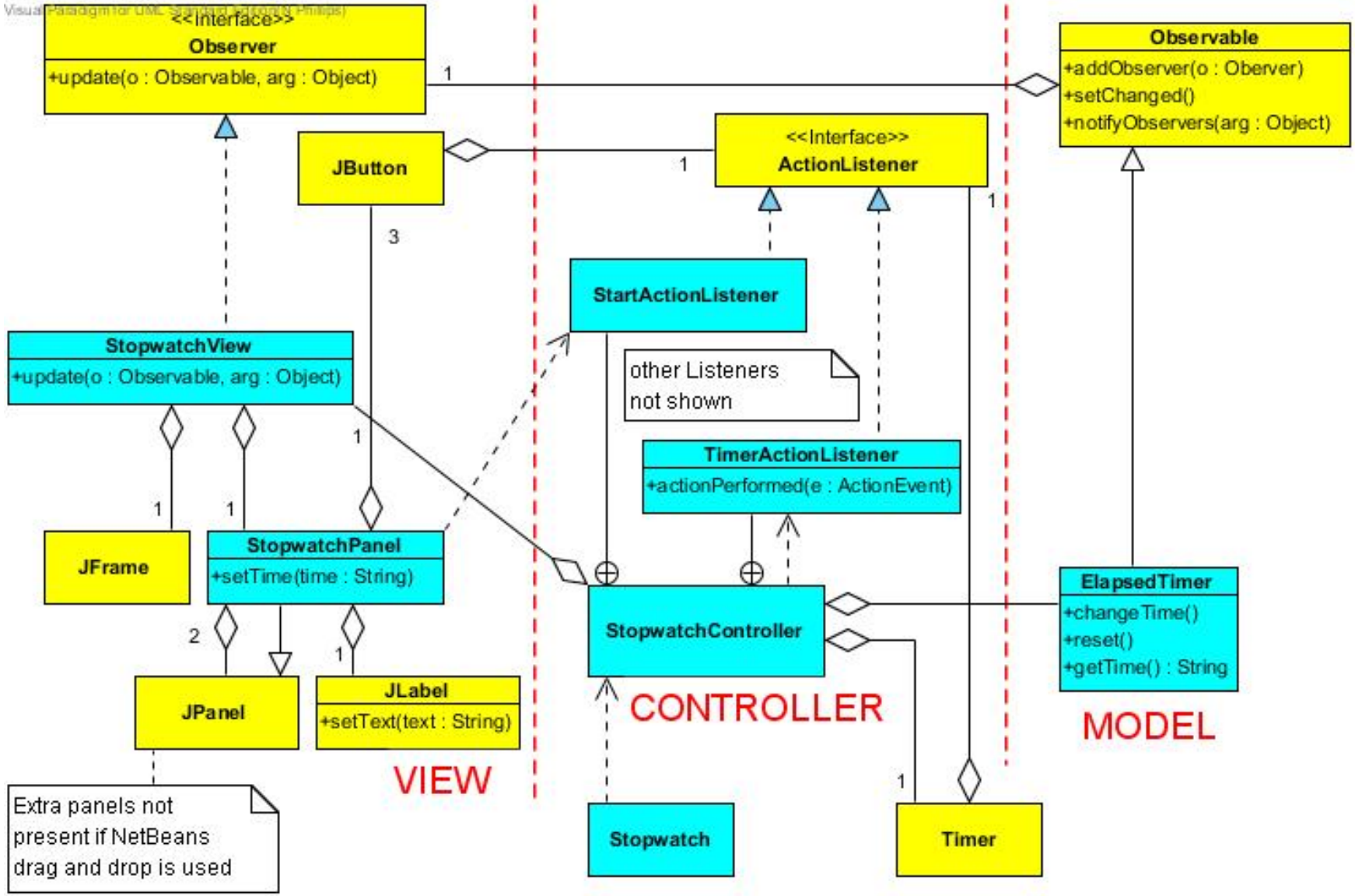
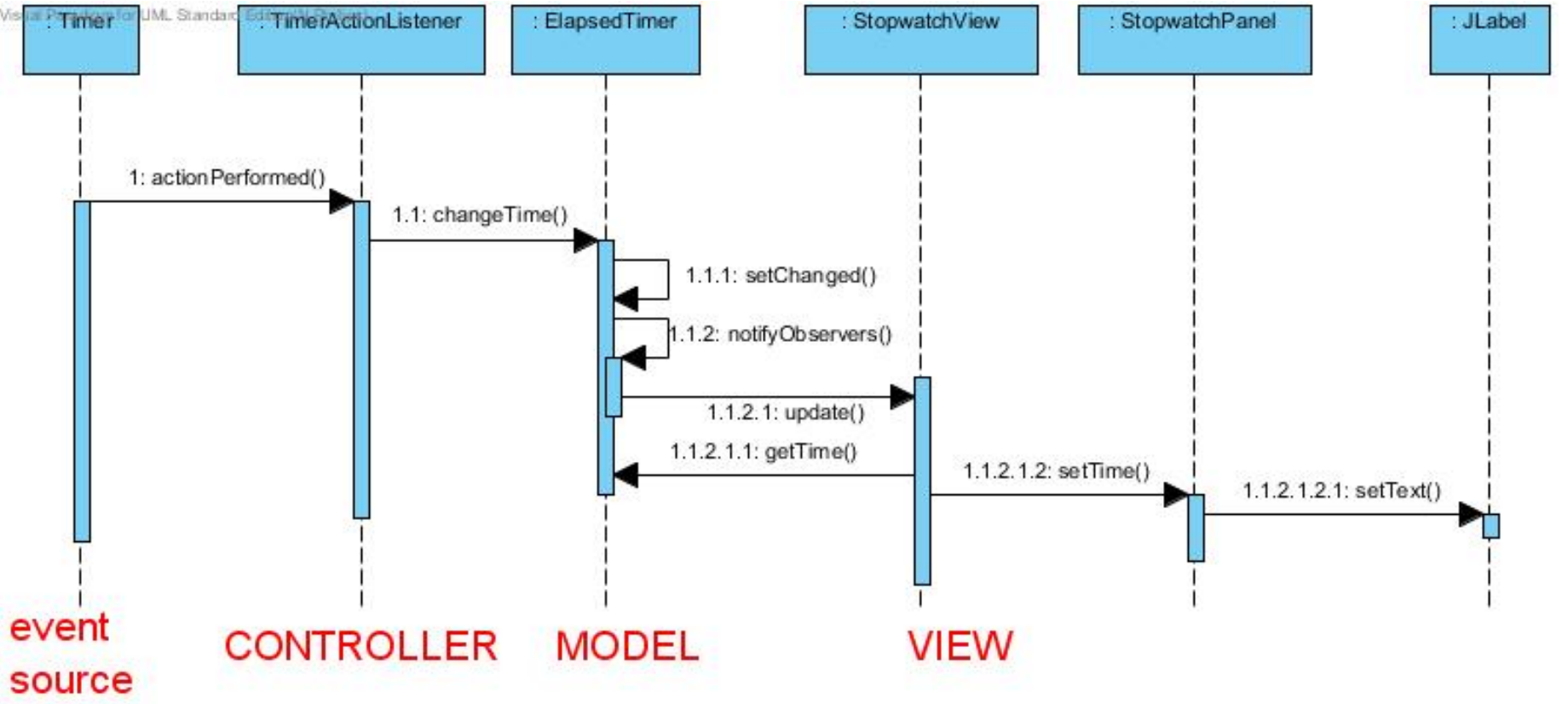
Lecture Week 5
The Stopwatch artefact (part 3)

The Stopwatch class diagram



Sequence when Timer ActionEvents are generated



ElapsedTimer

```
package stopwatch;
import java.util.Observable;
public class ElapsedTimer extends Observable {
    private long resetTime = 0;
    private int timeNow = 0;
    private static final int TENTHS_PER_SECOND = 10;
    private static final int SECONDS_PER_MINUTE = 60;
    private static final int SECONDS_PER_HOUR = 3600;
```

- The class extends Observable – which is part of the Java class library
  - This means it has available the methods *setChanged()* and *notifyObservers()* which can be called whenever a significant change occurs which its Observer(s) need to know about.
- The time *resetTime* is a *system time*. A system time is a time, measured in milliseconds, from midnight, January 1, 1970 – which is why it is a *long*. *resetTime* is the system time when the stopwatch has been reset to display 0, and *timeNow* is difference between the present system time and *resetTime* – i.e is the time in milliseconds that will be used for the display on the stopwatch
- The constants that are declared will be used for calculating the formatted version of *timeNow* as hours:minutes:seconds:tenths

ElapsedTimer *reset()*

```
public void reset() {
    resetTime = System.currentTimeMillis();
    timeNow = 0;
    this.setChanged();
    this.notifyObservers(Properties.TIME);
}
```

- The *reset()* method starts by setting *resetTime* to be the current time and then zeroizes *timeNow*
- It then calls *setChanged()* from the superclass Observable. This is an indicator that a significant change has happened, and that the next call to *notifyObservers(…)*, which follows immediately in this case, should call the *update(…)* method on the Observers that have been registered with the ElapsedTimer.
- The call to *notifyObservers(…)* has has as its parameter a constant String TIME (held in the class Properties) with the value 'time'. This parameter will be used in the *update(..)* method of the Observer(s) – in our case just the StopwatchView - to determine the type of change that has occurred.
- Once *notifyObservers(…)* has been called, *setChanged()* would need to be called again to ensure the next call to *notifyObservers()* will again call update() on its Observers

ElapsedTimer *changeTime()*

```
public void changeTime() {
    timeNow = (int)((System.currentTimeMillis()
            - resetTime));
    this.setChanged();
    this.notifyObservers(Properties.TIME);
}
```

- This method will be called every time a Timer event is generated in the running state of the stopwatch
- It obtains the current time via a static method call to *currentTimeMillis()* in the System class, and then subtracts *resetTime* putting the result into *timeNow*, so that *timeNow* contains the time in milliseconds to be formatted for display.
- As in the method *reset()* (see previous slide), the *setChanged()* and *notifyObservers(…)* methods are called in order for the the *update(…)* method to be called on all the registered Observers (in our case the StopwatchView).

ElapsedTimer *getTime()*

```
public String getTime() {
     long timeNowInTenths = timeNow/100;
     long seconds = (timeNowInTenths / TENTHS_PER_SECOND)
             % SECONDS_PER_MINUTE;
     long tenths = timeNowInTenths % TENTHS_PER_SECOND;
     long minutes = (timeNowInTenths
             / (TENTHS_PER_SECOND * SECONDS_PER_MINUTE))
             % SECONDS_PER_HOUR;
     long hours = timeNowInTenths
             / (TENTHS_PER_SECOND * SECONDS_PER_HOUR);
     String timeString = new String(hours + ":"
             + minutes + ":"
             + seconds + ":"
             + tenths);
     return timeString;
 }
```

- This method will get called by the update(…) method of StopwatchView
- It calculates and returns String of the form hours:minutes:seconds:tenths from the timeNow instance variable showing the stopwatch time in msecs.

StopwatchView – *update(…)*

```
public class StopwatchView implements Observer {
   . . .
   public void update(Observable observable, Object arg){
        if (arg.equals(Properties.TIME)) {
           stopwatchPanel.setTime(((ElapsedTimer)observable).getTime());
        }
   }
   . . .
}
```

- This method is called from the *notifyObservers()* method in the Observable (in our case this can currently only be the ElapsedTimer)
- The first parameter typed to Observable is a reference to the caller itself. The second parameter typed as an object, is the parameter originally supplied to *notifyObservers()* indicating the type of change that has happened.

StopwatchView – *update(…) cont'd*

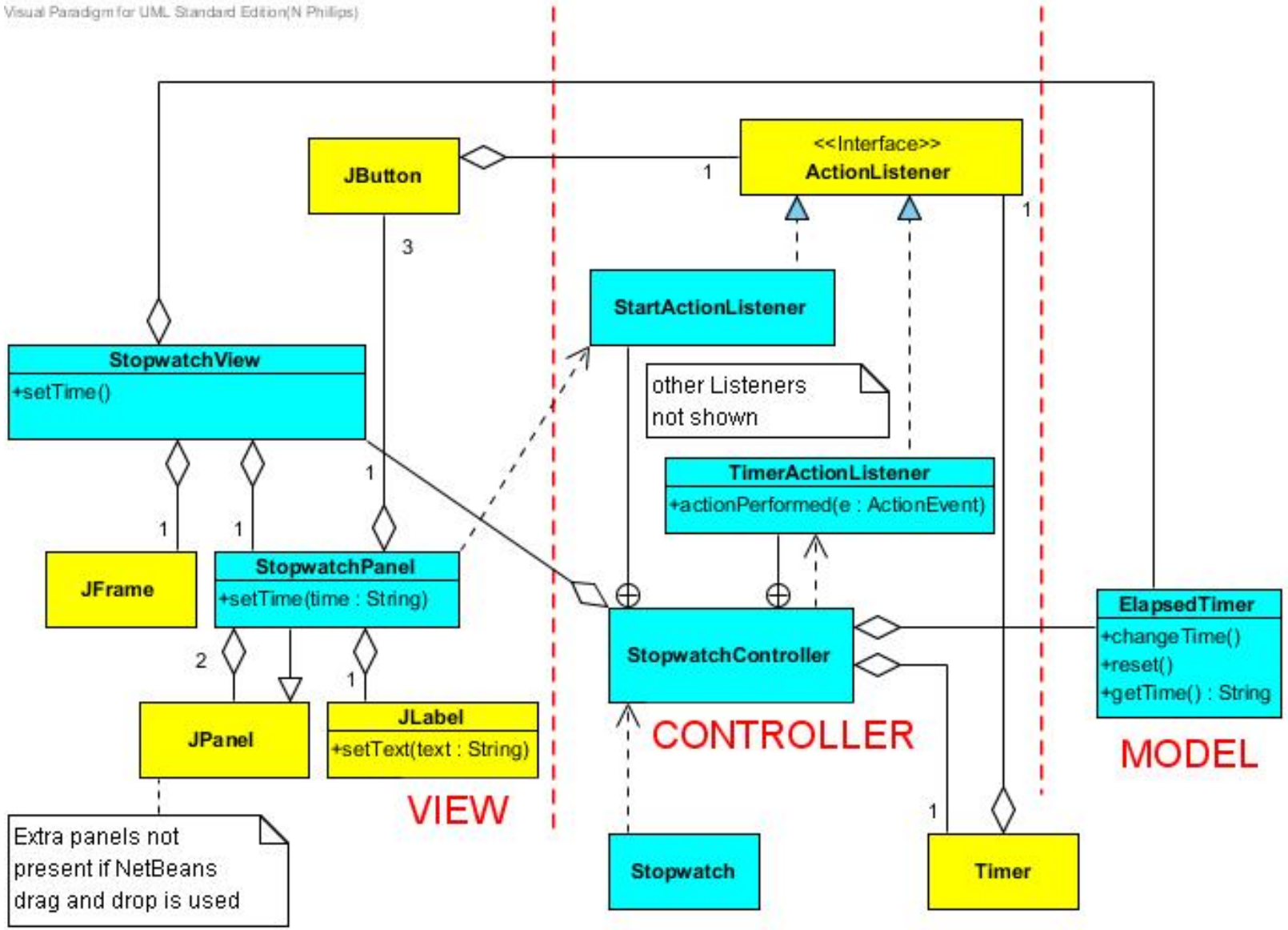
- The second parameter is checked for equality against the constant *Properties.TIME* (which has been given the value "time").
- If it is equal, observable needs to be cast to an Elapsed Timer and the method getTime() is called on it.
- The returned time value in the form of a String XX:XX:XX:XX is then directly passed as parameter to the *setTime(…)* method of StopwatchPanel which, as we saw, leads to the the JLabel for the time being updated.

The class Stopwatch

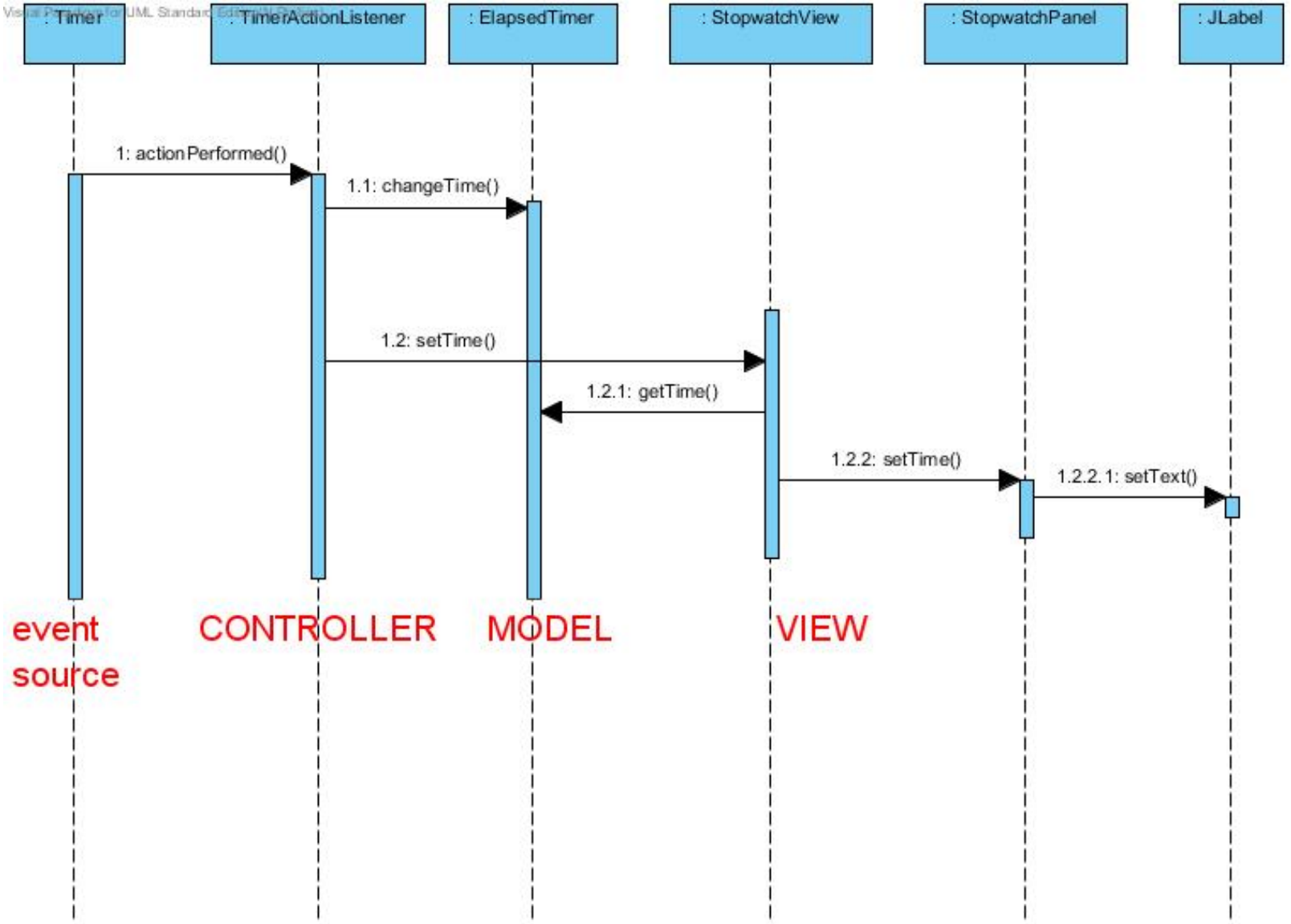
```
package stopwatch;
public class Stopwatch {
    . . .
    public static void main(String[] args) {
        StopwatchController sc = new StopwatchController();
        sc.start();
    }
}
```

- This class is used to start up the artefact
- The *main(…)* method constructs an instance of StopwatchController (which in turn initiates the calls to the constructors for the other classes) for the preparation phase
- It than calls the *start()* method on the StopwatchController which will effect the transition to the operation phase.

Passive MVC version of Stopwatch



Passive MVC version sequence diagram



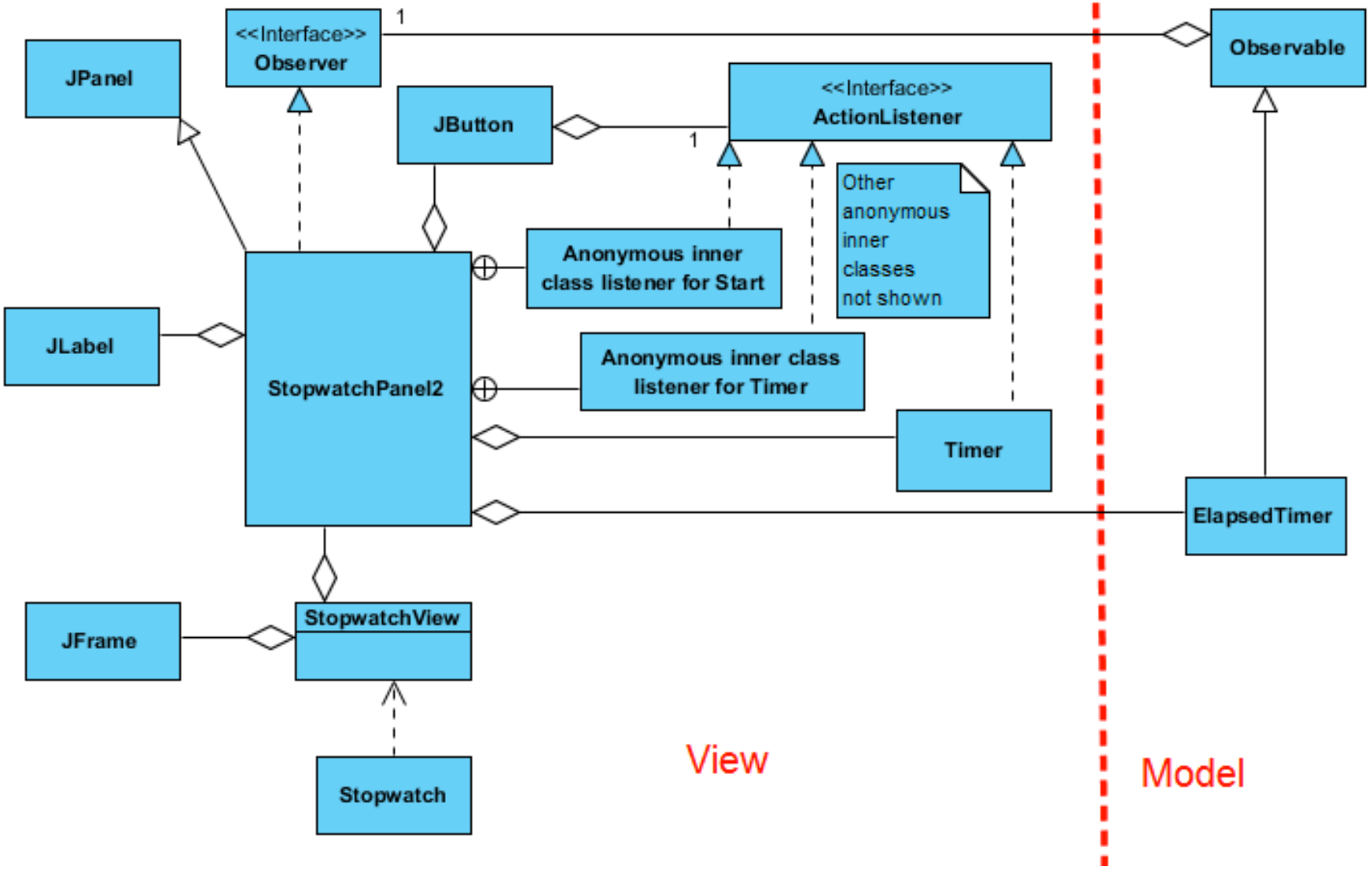
Stopwatch passive MVC – major points

- The passive MVC architecture differs from the active version in that the Model now does not inform the View(s) that it has changed. Data from the model must be "pulled" by a View since it is not "pushed" to its View(s), as it was in the active version.
  - This means it is harder to have multiple Views on a model or to swap one View of a Model with another View.
  - However it does tend to make it easier to trace the flow of control at run time and to carry out debugging
- In the previous class and sequence diagrams, it can be seen that in the passive MVC version of Stopwatch, the ElapsedTimer (i.e. the Model) is now not a subclass of Observable and the StopwatchView does not implement Observable. This means that the Observer pattern is now not in operation

Stopwatch passive MVC – major points cont'd

- The StopwatchView now aggregates, and is therefore coupled to the ElapsedTimer directly. Its constructor now needs an extra parameter of type ElapsedTimer (not shown on class diagram for space reasons) in order for this aggregation to be set up.
- The Controller now after calling *changeTime()* on the ElapsedTimer, then calls *setTime()*, (now a method with no parameters) on StopwatchView.
- *setTime()* of StopwatchView then gets the formatted version of the time directly from ElapsedTimer by calling its *getTime()* method

Stopwatch Two Layer Architecture



Stopwatch Two Layer architecture cont'd

- The previous slide shows class diagram for the stopwatch artefact with the controller removed
- This is now a Model View architecture as opposed to a Model View Controller Architecture
- However the Observer Pattern is still used with changes in the ElapsedTimer being reflected in the display of the time via the *notifyObservers()/update()* mechanism
- The StopwatchPanel2 class now aggregates the Timer and ElapsedTimer directly – before these were aggregated by StopwatchController. Also now it is StopwatchPanel (rather than StopwatchView) that now implements Observer
- The implementation has used the NetBeans GUI to both draw the layout using drag and drop and to auto generate the listener anonymous inner classes
  - You are asked *not* to use the NetBeans GUI to auto-generate listener anonymous inner class for your coursework!!

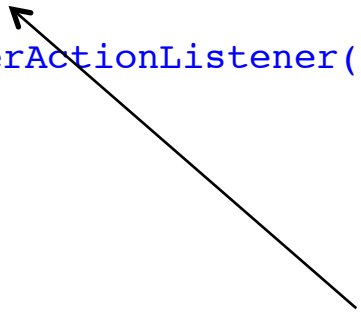
Two Layer architecture StopwatchPanel2

```
public class StopwatchPanel2 extends javax.swing.JPanel implements Observer {

    private final static int RESET = 1;
    private final static int RUNNING = 2;
    private final static int STOPPED = 3;

    //The current state
    private int theState = RESET;
    //The timer
    private Timer aTimer;
    // The Model - the object supplying the stopwatch time
    private ElapsedTimer elapsedTimer;
    . . .
}
```

- StopwatchPanel2, instead of StopwatchController now aggregates the ElapsedTimer and the Timer
- Instead of StopwatchController it also manages the state machine
- It also acts as the Observer for changes in the Observable ElapsedTimer

Two Layer Architecture StopwatchPanel2 cont'd

```
public class StopwatchPanel2 extends javax.swing.JPanel implements Observer
  . . .
  public StopwatchPanel2() {
     initComponents();//execute initialization code generated by drag and drop
     elapsedTimer = new ElapsedTimer();
     elapsedTimer.addObserver(this);
     aTimer = new Timer(50, new TimerActionListener());
  }
  . . .
}
```



- The StopwatchPanel2 is added as an Observer to ElapsedTimer

Two Layer Architecture StopwatchPanel2 cont'd

```
public class StopwatchPanel2 extends javax.swing.JPanel implements Observer {
    . . .
    @Override
    public void update(Observable observable, Object arg){
        if (arg.equals(Properties.TIME)) {
            this.setTime(((ElapsedTimer)observable).getTime());
        }
    }
    . . .
}
```

- Because it is now implements Observer, StopwatchPanel2 now has the *update()* method

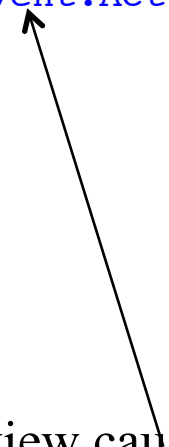
Two Layer Architecture StopwatchPanel2 cont'd

```
public class StopwatchPanel2 extends javax.swing.JPanel implements Observer {
    . . .
    public class TimerActionListener implements ActionListener {
        @Override
        /**
         * If in running state, ask ElapsedTimer to change time
         * (will then propagate change to the View to update the display)
         * @param event the ActionEvent from the Timer triggering the listener
         */
        public void actionPerformed(ActionEvent event) {
            if (theState == RUNNING){
                elapsedTimer.changeTime();
            }
        }
    }
    . . .
}
```

- TimerActionListener is now an inner class of StopwatchPanel2

Two Layer Architecture StopwatchPanel2 cont'd

```
public class StopwatchPanel2 extends javax.swing.JPanel implements Observer {
    . . .
    private void initComponents() {
        startButton = new javax.swing.JButton();
        . . .
        startButton.setText("Start");
        startButton.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                startButtonActionPerformed(evt);
            }
        });
        . . .
     }
     . . .
}
```



- Double clicking on the startButton in the design view causes an anonymous – i.e. unnamed inner class which is then added as its ActionListener, with its required *actionPerformed()* method – note the new and rather strange syntax
- *actionPerformed()* merely contains a call to the method *startButtonActionPerformed()* in the outer class StopwatchPanel2

Two Layer Architecture StopwatchPanel2 cont'd

```
public class StopwatchPanel2 extends javax.swing.JPanel implements Observer {
    . . .
   private void startButtonActionPerformed(java.awt.event.ActionEvent evt) {
            if (theState == RESET) {
                elapsedTimer.reset();
                theState = RUNNING;
                setButtons(false,true,false);
            }
     }
     . . .
}
```

- This method carries out the code to implement the transition and corresponding action between the reset state and running state (previously in *actionPerformed()* method of the inner class StartActionListener within Stopwatch Controller).
- NetBeans jumps to the point at which the developer fills in this content..

Model-View versus Model View Controller

- For more complex systems than Stopwatch, using a Model-View as opposed to Model View Controller architecture involves severe bloating of code in a class such as, in this example, StopwatchPanel2
- It also involves mixing up the control aspects of the system with the visual display. This means that
  - it becomes impossible to separately change each of these aspects
  - it is harder to trace bugs
  - It is harder for different developers to work on these aspects simultaneously
- Specifying event handling interactively using a GUI such as NetBeans results in code bloat in presentation classes and a reduction of engineering rigour in the resulting auto-generated code

Refactoring

- Moving from Model-View (MV) to Model View Controller (MVC) architecture is an example of *refactoring* (you should read Chapter 13 of the recommended text book Software Engineering for Students by Bell)
- Refactoring reorganises code in order to improve architecture and the engineering quality of a software system. Overall functionality is unaltered.
- It can involve several forms of reorganisation
  - Encapsulation into a class of previously public shared data
  - Moving a method from one class to another
  - Moving a field from one class to another
  - Creating a new class
    - in our case refactoring from MV to MVC involves creating the controller class)
  - Identifying composition or inheritance in order to promote software re-use
    - For example identifying common fields and behaviour for a top level product class and subclassing for specialised products