Lecture Week 4
The Stopwatch artefact (part 2)

The Observer Design Pattern

- "The **observer pattern** is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems. The Observer pattern is also a key part in the familiar Model View Controller (MVC) architectural pattern. [1] In fact the observer pattern was first implemented in Smalltalk's MVC based user interface framework.[2] The observer pattern is implemented in numerous programming libraries and systems, including almost all GUI toolkits."

  - Wikipedia (look up this link for a generalised class diagram)
- The relationship between listeners and event sources in Java is an example of the Observer Pattern.
- The pattern is one of many from the landmark Book of 1995 "Design Patterns, Elements of Re-usable Object Oriented Software" by Gamma, Helm, Johnson and Vlissides)

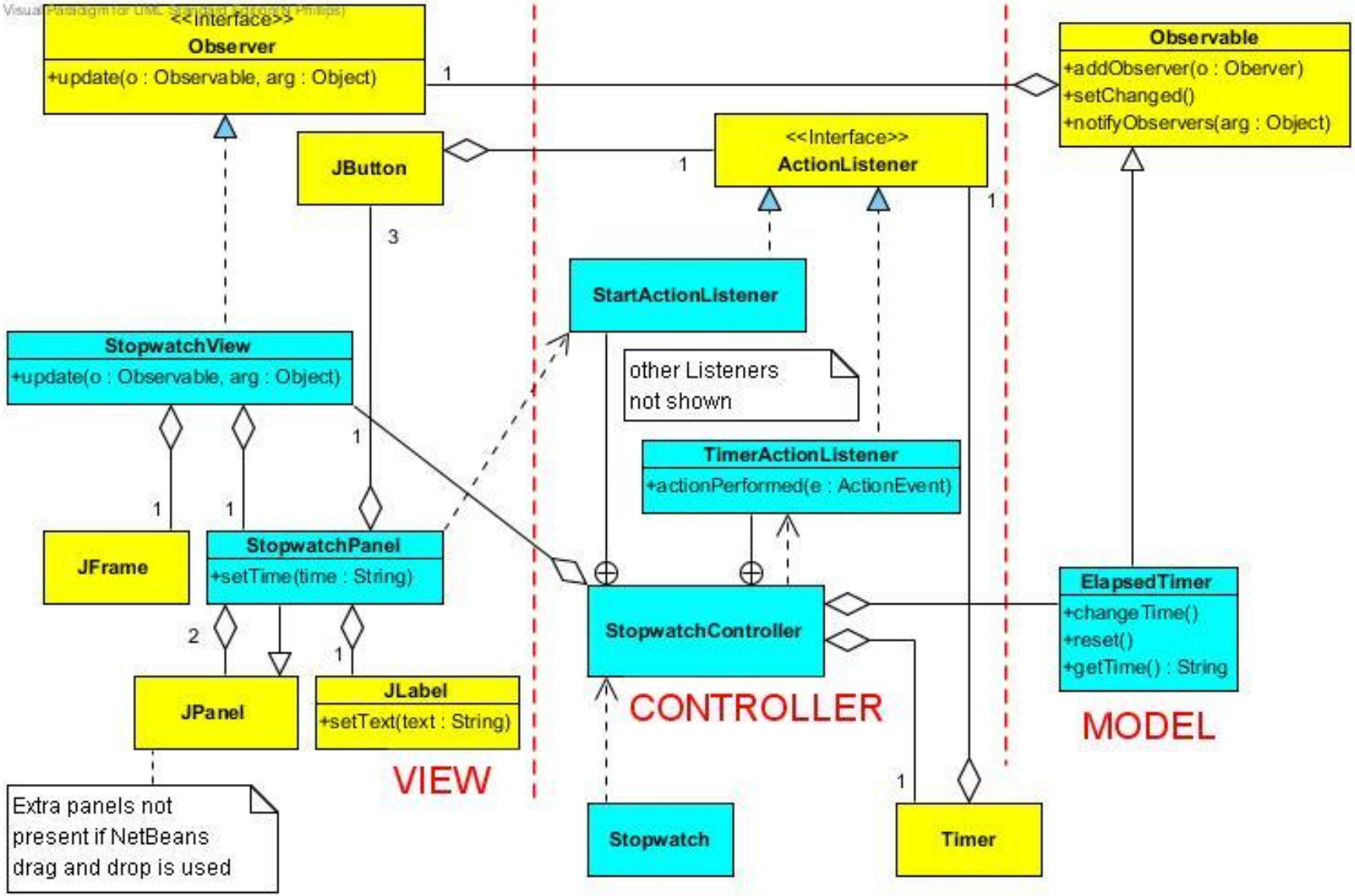
More on Model View Controller Architecture

- Reference:
  - Microsoft paper on MVC (with further references)
- Separation concerns between software components is vital for software maintainability and testing
- The Model can exist alone, outside of interactive system, so that independent full testing can take place (using for example JUnit in Java)
- User interface software tends to change more frequently than the model software and allows different user interfaces to be plugged in to the same model.
  - Either one view replacing another or several views operating simultaneously

More on MVC architecture 2

- There are two variants to the MVC architecture – passive and active
- In the *passive* variant the Controller receives events resulting from user actions and potentially both updates or queries the Model and updates the View. The web application use of MVC follows this variant
- The *active* variant of MVC involves the Observer pattern and allows for the possibility the Controller updating the Model, and then the Model itself updating its associated View or Views. The Model only sees its views as being Observers via a publish-subscribe mechanism. This gives loose coupling and *another* case of the use of the Observer pattern.
  - in addition to the case of the relationship between event source and Listener
- In our examples we use the active variant of MVC
- In our examples the Controller embodies the logic of the State Transition Diagram

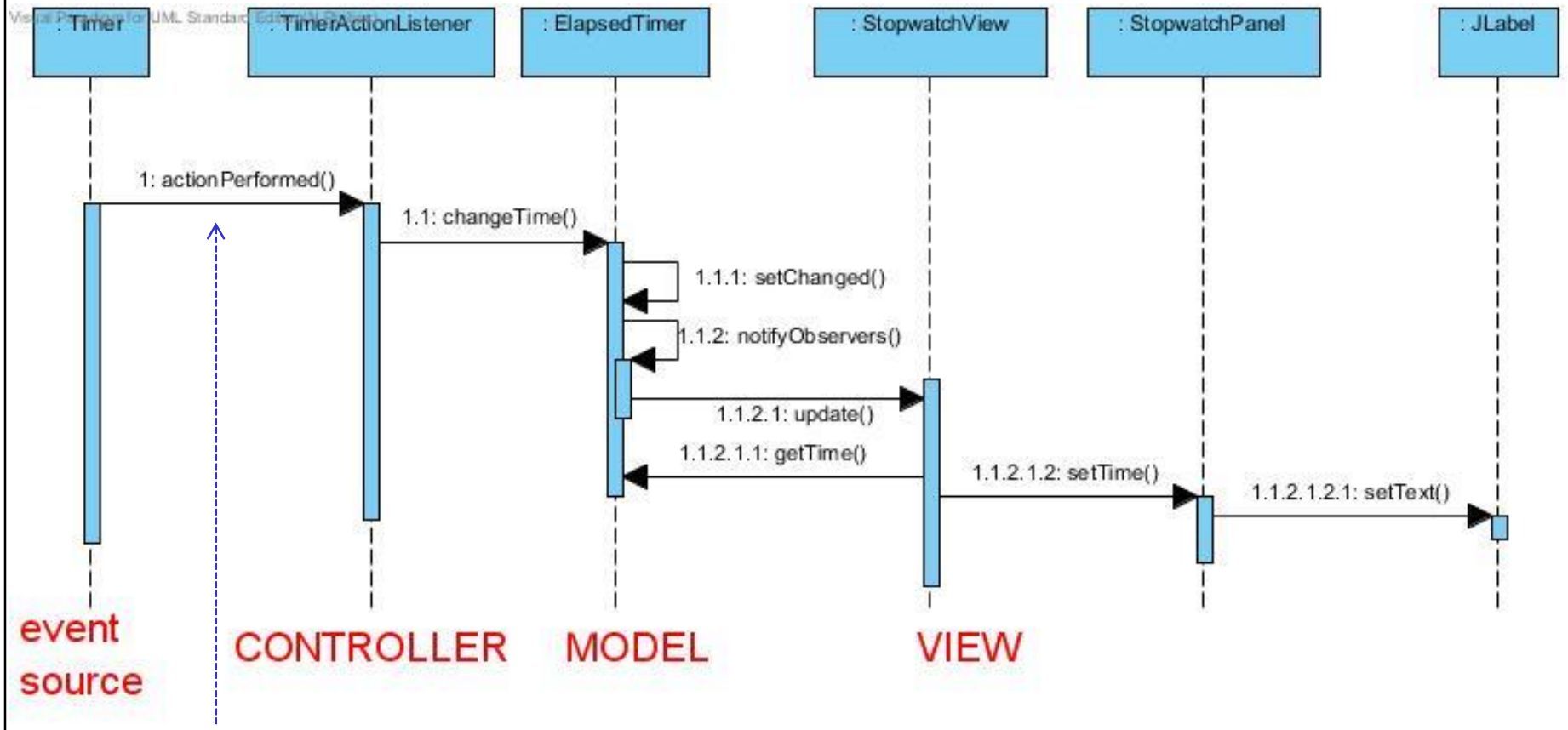
The Stopwatch class diagram



MVC in our example

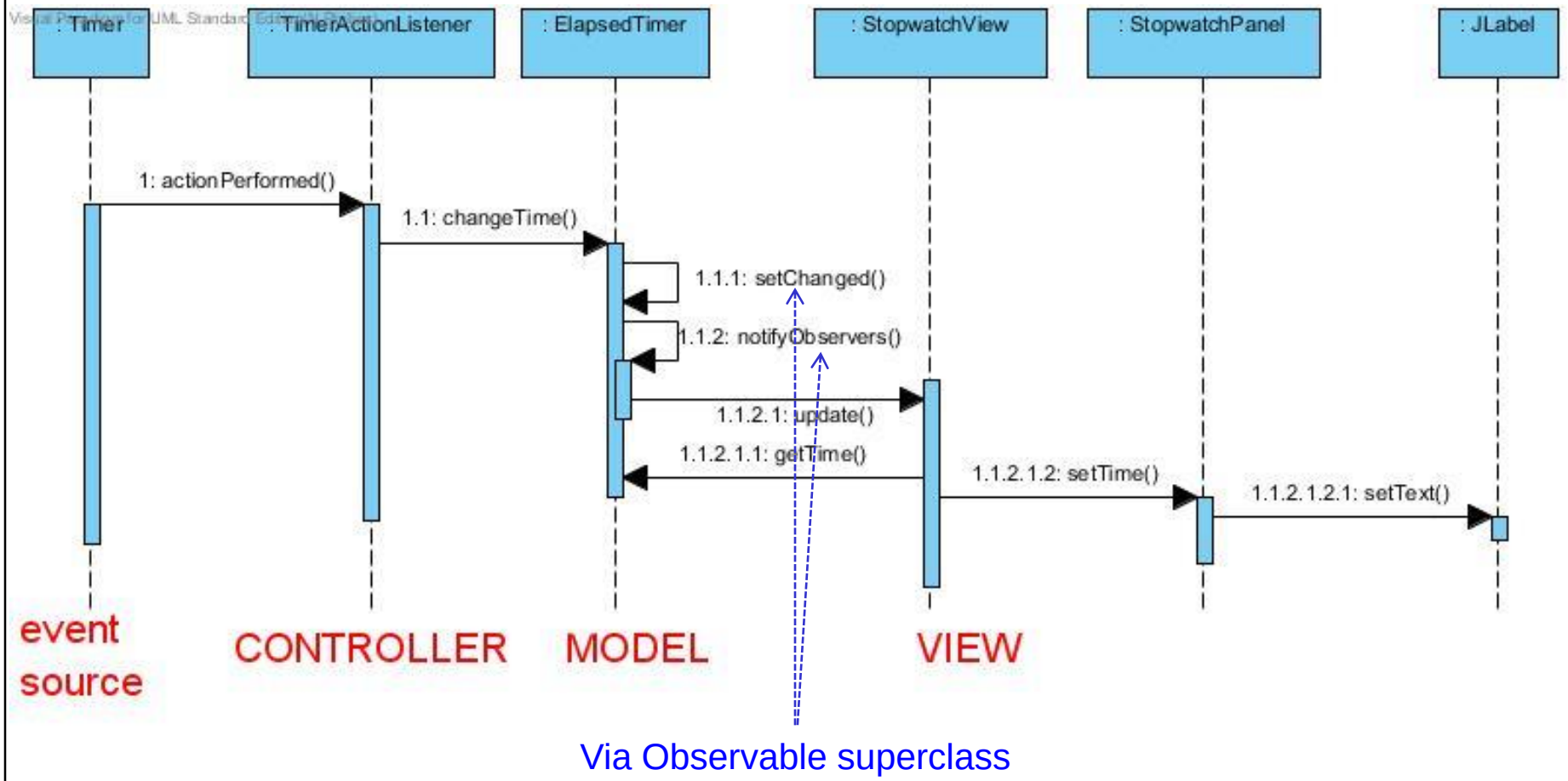
- ElapsedTimer is a subclass of Observable from the Java class library. Observable aggregates a set of Observers (in our case only one). Observer is an interface in the Java class library
- The class StopwatchView implements the interface Observer
- In the preparation phase StopwatchController adds StopwatchView to the list of Observers of the ElapsedTimer by calling the *addObserver(…)* method on it.
- In the operation phase, whenever the Timer emits an event, it calls *actionPerformed(…)* on its listener which in turn, in the running state, calls *changeTime()* on the ElapsedTimer
- This in turn, via calls to the superclass methods *setChanged()* and *notifyObservers()*, results in the calling of method *update(…)* on all its Observers (in this case only one – the StopwatchView)
- The StopwatchView then requests the current time data from the ElapsedTimer and displays the data on its StopwatchPanel.

Sequence when Timer ActionEvents are generated

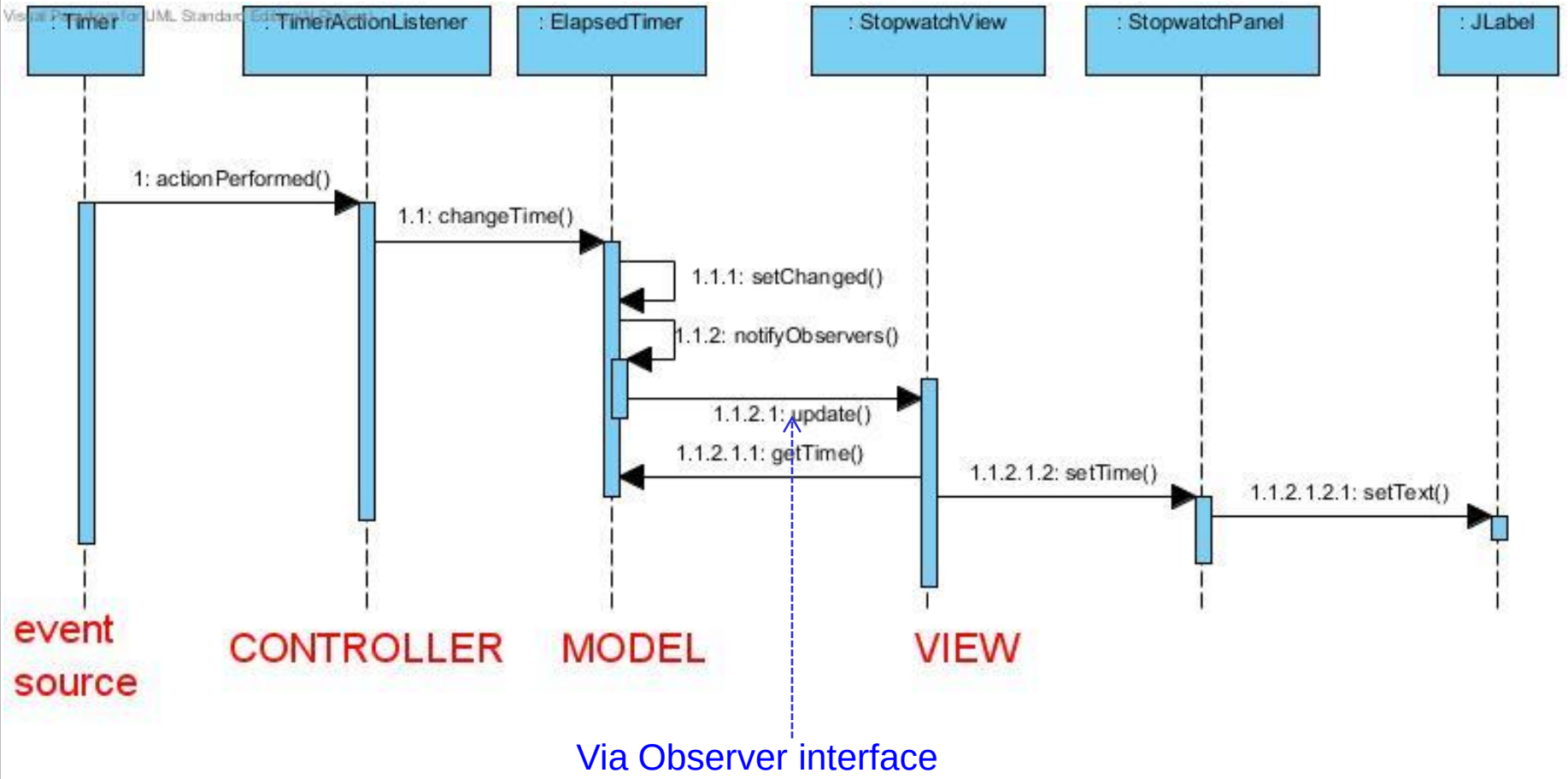


Via ActionListener interface

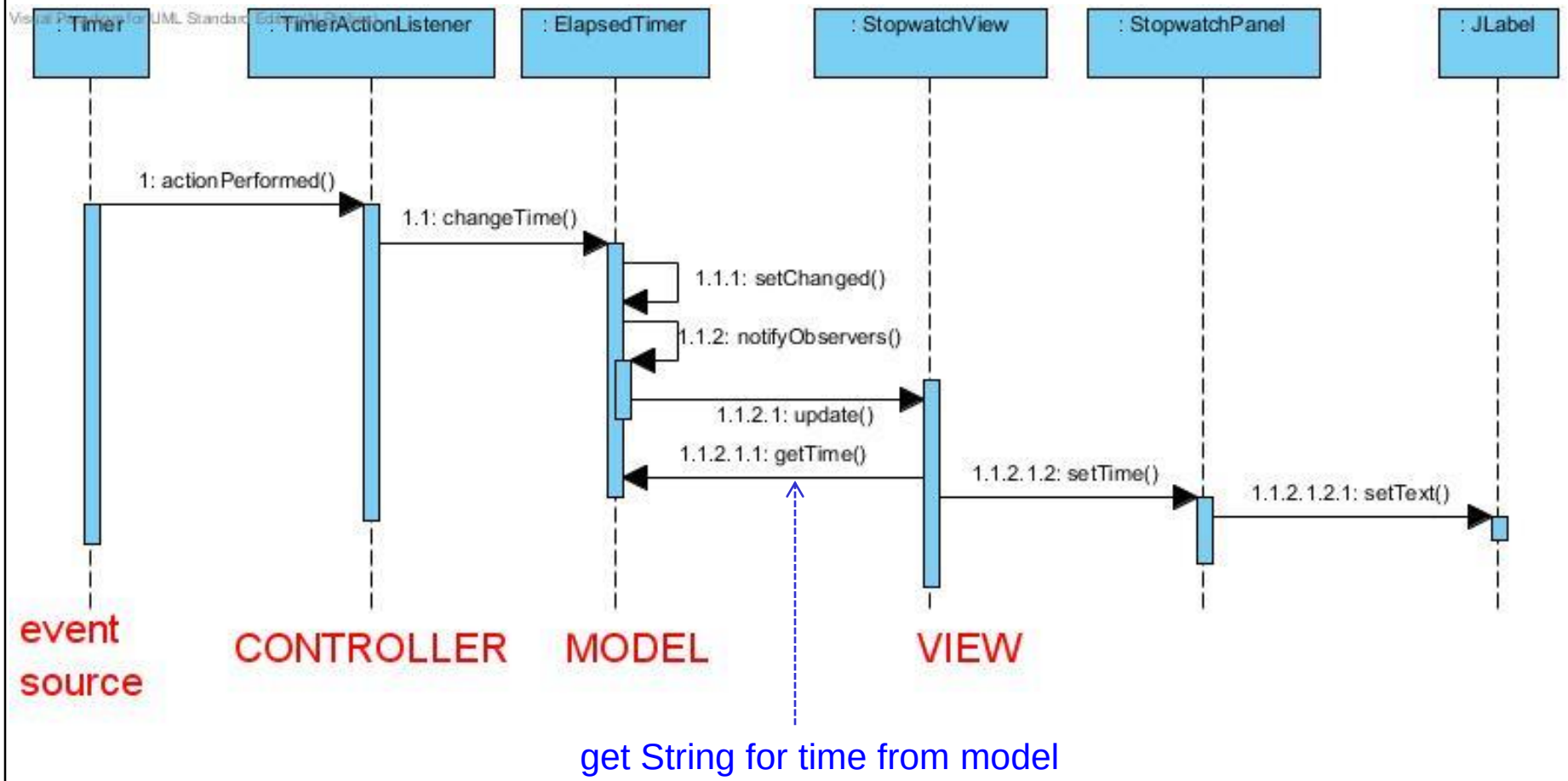
Sequence when Timer ActionEvents are generated



Sequence when Timer ActionEvents are generated



Sequence when Timer ActionEvents are generated



Sequence diagrams

- The previous diagrams are UML *sequence diagrams*
- More information on these can be obtained from Visual Paradigm and IBM. Visit these links to extract enough information to understand the sequence diagrams in these lectures.
- In the previous diagrams the lifelines were labelled for example :Timer , :TimerActionListener etc. These indicate instances of the class Timer, TimerActionListener etc. The names of the instances normally come before the ‘:’, but leaving them out means ‘an instance of’

StopwatchView

```
package stopwatch;
import java.awt.Color;
import java.awt.Font;
import java.util.Observable;
import java.util.Observer;
import javax.swing.JFrame;
import javax.swing.UIManager;
import javax.swing.plaf.ColorUIResource;
import javax.swing.plaf.FontUIResource;

public class StopwatchView implements Observer {
    private StopwatchPanel stopwatchPanel;
    //private StopwatchPanel2 stopwatchPanel;
    private JFrame frame;
```

- After declaring the package and imports, the declaration of the class shows that it implements the Observer interface, so the class will have to implement *update(..)*
- The two instance variables *stopwatchPanel* (of type StopwatchPanel) and *frame* (of type JFrame) are then declared

StopwatchView - Constructor

```
. . .
public StopwatchView(StopwatchController controller) {
        super();
        this.setResources();
        frame = new JFrame("Stopwatch");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        stopwatchPanel = new StopwatchPanel(controller);
        frame.add(stopwatchPanel);
        frame.pack();
}
. . .
```

- The constructor gets passed a reference to the StopwatchController.
- The call to private method *setResources()* sets up the visual properties for fonts and Swing components (to be covered later)
- A new instance of StopwatchPanel, based on the controller passed in, is then constructed
- It is then added to the frame, and the frame is *pack()*ed so that it is the right size to enclose the StopwatchPanel

This code is part of the Preparation Phase

StopwatchView - methods

```
. . .
    @Override
    public void update(Observable observable, Object arg){
         . . .
    }

    public void show() {
        frame.setVisible(true);
    }
    . . .
```

- The *update(. . .)* method is part of the implementation of the observer pattern and will be dealt with later
  - This method is part of the Operation phase
- *The show()* method is called when display of the interface and the the starting of the Swing event loop is required.
  - The single line in the body of this method makes this happen by making the JFrame and all of its subpanels and low level components within them appear to the user
  - This method is part of the transition from the Preparation phase to the Operation Phase

StopwatchView – methods (cont'd)

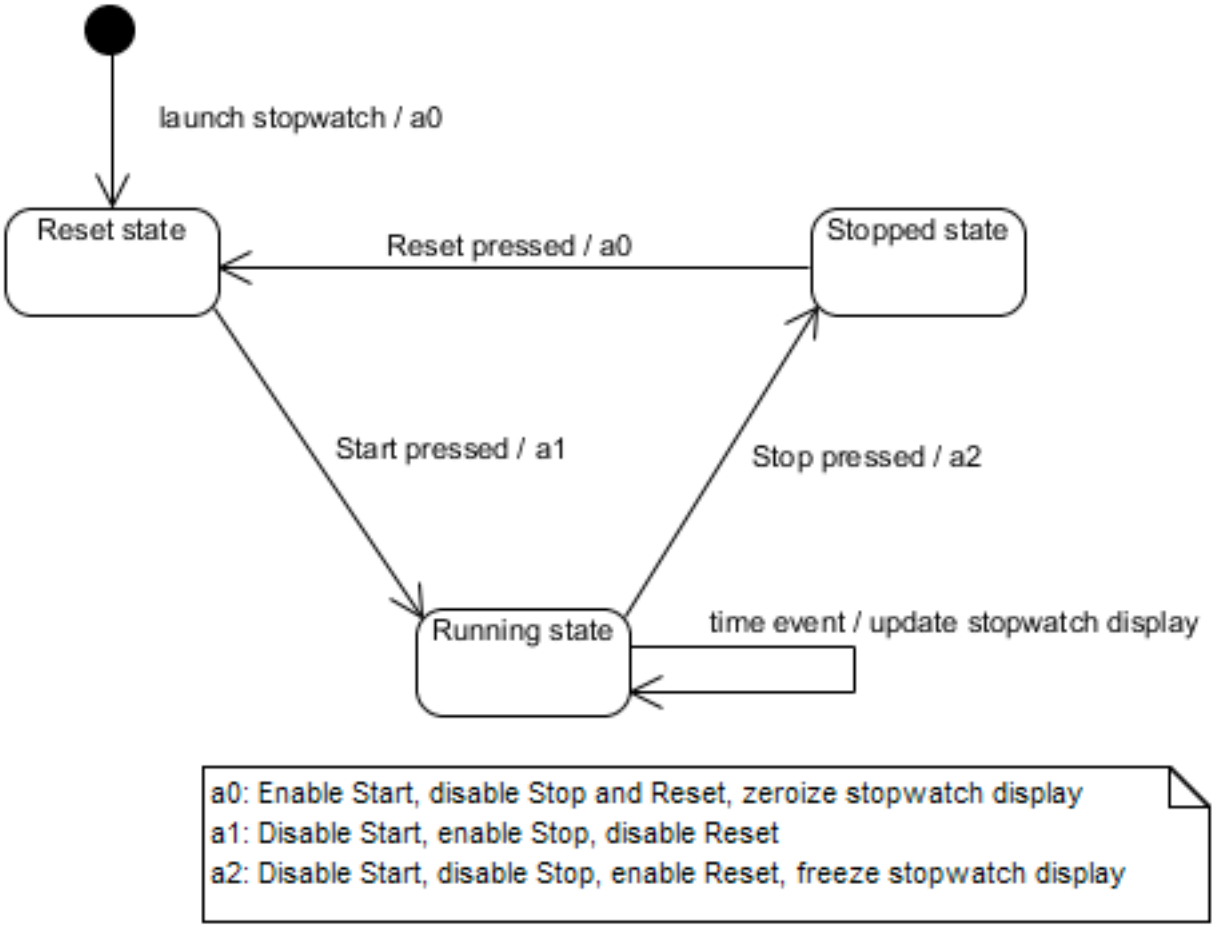
```
public void setResetState() {
    stopwatchPanel.setButtons(true,false,false);
}

public void setRunningState() {
    stopwatchPanel.setButtons(false,true,false);
}

public void setStoppedState() {
    stopwatchPanel.setButtons(false,false,true);
}
```

- The first of these methods asks the StopwatchPanel to set the Start Button to be enabled, and the Stop and Reset buttons to be disabled
- The other methods invoke *setButtons(. . .)* on the StopwatchPanel to set the appropriate button enabling/disabling for the other states.

These methods are part of the Operation Phase, although *setResetState()* is also called in the Preparation Phase

The Stopwatch artefact
- visual appearance

Stopwatch State Diagram



StopwatchController

```
package stopwatch;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.Timer;

public class StopwatchController {
    private final static int RESET = 1;
    private final static int RUNNING = 2;
    private final static int STOPPED = 3;
    private int theState = RESET;
    private Timer aTimer;
    private ElapsedTimer elapsedTimer;
    private StopwatchView theView;
```

- After the package and import declarations, the class declaration constants representing the different states of the state transition diagram are declared and an instance variable *theState* which holds the value of the current state (and will change) is also declared
- Instance variables used to reference the Timer, ElapsedTimer and StopwatchView are also declared

StopwatchController – constructor

```
public StopwatchController() {
    elapsedTimer = new ElapsedTimer();
    theView = new StopwatchView(this);
    theView.setResetState();
    elapsedTimer.addObserver(theView);
    aTimer = new Timer(50, new TimerActionListener());
}
```

- The constructor creates the instance of ElapsedTimer
- It creates the instance of StopwatchView passing it the parameter *this* to indicate that (eventually) events from the buttons will be passed to the inner class listeners of the this StopwatchController
- The View is then set to the reset set (which will ask the StopwatchPanel to set its buttons appropriately)
- The ElapsedTimer will be given the StopwatchView as an Observer via the *addObserver(…)* method call
- The Timer is then constructed with a 50 millisecond period whose ActionListener will be a newly constructed TimerActionListener (inner class)

This code is part of the Preparation Phase

StopwatchController - *start( )*

```
. . .
public void start() {
    theView.show();
    elapsedTimer.reset();
    aTimer.start();
}
. . .
```

- This method is called to kick off the artefact (after all of its components have been constructed)
  - Calling *show()* on the StopwatchView instance *theView* will cause the enclosing JFrame to be be made visible and the event loop waiting for user events to be started
  - Calling reset() on the ElapsedTimer will cause the model to be reset to 0 causing the View to display 0:0:0:0
  - Calling *start()* on the Timer instance *aTimer* will cause the clock to start issuing ActionEvents

This code has the transition from the Preparation Phase to the Operation Phase

StopwatchController - StartActionListener

```
public class StopwatchController
    . . .
    public class StartActionListener implements ActionListener {
         public void actionPerformed(ActionEvent event) {
             if (theState == RESET) {
                 elapsedTimer.reset();
                 theState = RUNNING;
                 theView.setRunningState();
             }
         }
     }
     . . .
}
```

•StartActionListener is an inner class of StopwatchController. It implements ActionListener. Its *actionPerformed(…)* method will be called when the user clicks the Start button. An inner class has access to fields and methods of the enclosing class.

•If the current state is RESET, the method calls *reset()* on the ElapsedTimer, (which will cause the model to be reset to 0 causing the StopwatchView to display 0:0:0:0). It then changes the state to RUNNING according to the state transition diagram, and calls *setRunningState()* on the StopwatchView.

This code is part of the Operation Phase

StopwatchController - StopActionListener

```
public class StopwatchController
    . . .
    public class StopActionListener implements ActionListener {
         public void actionPerformed(ActionEvent event) {
             if (theState == RUNNING) {
                 theState = STOPPED;
                 theView.setStoppedState();
             }
         }
     }
     . . .
}
```

- StopActionListener is another inner class of StopwatchController
- *actionPerformed(…)* executed when the Stop button is clicked
- It the current state is RUNNING when the Stop button is pressed, the state is changed to STOPPED and *setStoppedState()* is called on the StopwatchView to cause the button sensitivities to reflect the new state

This code is part of the Operation Phase

StopwatchController - ResetActionListener

```
public class StopwatchController
    . . .
    public class ResetActionListener implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            if (theState == STOPPED) {
                theState = RESET;
                elapsedTimer.reset();
                theView.setResetState();
            }
        }
    }
. . .
}
```

- ResetActionListener is another inner class of StopwatchController
- *actionPerformed(…)* executed when Reset button is clicked
- If the current state is STOPPED, then in addition to changing the state to RESET and getting *theView* to reflect the new state, the ElapsedTimer method *reset()* is called, which will cause the model to be reset to 0, causing the StopwatchView to display 0:0:0:0.

This code is part of the Operation Phase

StopwatchController -TimerActionListener

```
public class StopwatchController {
    . . .
    public class TimerActionListener implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent event) {
            if (theState == RUNNING){
                elapsedTimer.changeTime();
            }
        }
    }
    . . .
}
```

- TimerActionListener is another inner class of StopwatchController
- *actionPerformed()* executed when the Timer issues an ActionEvent
- If the state is RUNNING then *changeTime()* is called on the ElapsedTimer. This will cause the ElapsedTimer to compute the current time for stopwatch and propagate it for display to the StopwatchView

This code is part of the Operation Phase