Chapter 3

Software Engineering Event Based Systems

3.1 Introduction

As a student of computer science you have probably written a program at least 1,000 - 2,000 lines long. If you are a good student (and we know you are), you undoubtedly sat and thought about the program for quite a while before you started coding. Do you remember what those thoughts were?

The authors are a couple of computer science professors. Every semester we have one or two students who relish the personal attention they receive during office hours. One of our personal favorites was Thor R. Roscoe. Every time we gave an assignment, Thor behaved exactly the same way. He would wait until the day before the assignment was due, and then spend 24 hours straight trying to get it completed on time.

First, he'd read the assignment. Then he'd come straight to our office and say, "I'm completely lost. I have no idea how to begin. What's this project supposed to do?" We'd patiently tell him that he should have come to see us two weeks earlier, when we initially assigned the project. Then we'd (still patiently, but perhaps a bit less so) explain to him the intention of the assignment, and send him on his way.

Thor would go down to the computer lab, scratch his head for a few hours more, and then come back up. "Okay, I think I know what the program is supposed to do, but I'm still completely lost. How do I go about programming it?" We'd sit and draw some class diagrams with him, or explain what data structures he needed (generally the ones being studied in class), and send him on his way – again.

Thor would go back down to the lab and spend all night hacking code. He'd write some wonderful code, sometimes correct, sometimes incorrect, but always wonderfully indented and commented, since he knew he could get some points just for nice looking code. By morning, he'd be back, waiting for us outside our offices when we arrived, with just one question, "Is this right?" Our answer was always the same, "Turn it in when you are ready and we will give it a grade."

Thor would submit his assignment, then go back to his dorm room and sleep through class [1].

Belief it or not, Thor was a natural software engineer. He knew when he didn't understand something, and he knew the correct questions to ask. What's the system supposed to do? How is it going to accomplish it? How do we verify that it is implemented correctly? Finding and documenting answers to these questions forms the heart of software engineering.

[1] Sometimes he would show up, but fall asleep during class, which was even less preferable.

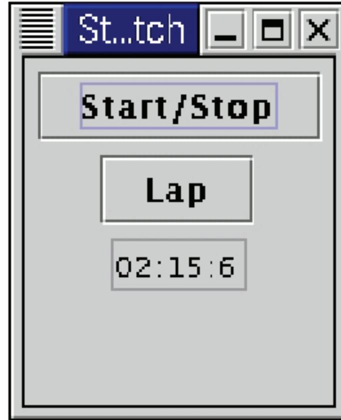

Figure 3.1: A simple graphical user interface for a two button stopwatch.

This chapter explores answering these questions for event based systems. As we have already seen, events have broad applicability in computing. While these computing fields are diverse, their event based nature lets us apply many of the same software engineering techniques to all of them. Also, as we have discussed earlier, very few programs are purely event based. Most programs use events where appropriate and use procedural or object-oriented techniques elsewhere. We concentrate on just those techniques that are particularly applicable to the event based portion of the system.

3.2 The Two Button Stopwatch

A good way to approach the development of an event based system is to walk through a single example from start to finish. This chapter follows the development of a program to simulate a stopwatch. We chose this example because it is relatively simple, yet clearly illustrates many of the methods and techniques of interest.

A stopwatch is used to time athletes. It typically has accuracy down to tenths of seconds. One standard model of a stopwatch has a digital display to show the elapsed time and two buttons to control its operation. A user interface for a simulation of this type of watch is shown in Figure 3.1.

The *Start/Stop* button starts and stops the watch.

The *Lap* button has more complex behavior. It lets the user freeze the display, so she can measure ”split” times without actually stopping the watch. For example, during a relay multiple athletes compete sequentially. When the first athlete completes their portion, the second one starts. Coaches are interested in the overall performance of the relay team, but are also interested in the performance of each individual. Clicking the lap button when one athlete finishes freezes the watch's display and lets the coach accurately record the elapsed time, while the stopwatch continues to run in the background, so the time for the next athlete and the overall relay time can also be obtained. Clicking the lap button a second time unfreezes and updates the display so that it again shows the elapsed time.

If the watch is stopped, clicking the lap button resets the time to 0.0.

If the watch is stopped while the display is frozen, clicking the lap button updates the display to the final elapsed time. A second click is necessary to reset the watch.

The stopwatch is a simple, but fairly typical event based system. It is characterized by being in a stable control state that changes only when events occur. It uses external events, button clicks, for user interaction. While running, it uses an internal timer (generating events) to update the

elapsed time, and as we will see later in the chapter, it may use other internal events to facilitate the communication among the parts of the system.

3.3 Event Based Software Development

A *software development methodology* is a disciplined approach imposed on the process of developing software. Using a well defined methodology has repeatedly been shown to increase productivity and project success. If developers better understand what the software is to do, and what their development responsibilities are, much less time is wasted and many fewer bugs are introduced. Many examples in this text are intentionally short, to illustrate some particular idea, but in the "real" world projects are much larger, often having teams of 50 or more programmers working on a single system. In this case, a well defined software development methodology becomes even more critical.

A software development methodology addresses two questions:

1. What is the development process, e.g. what are the steps used to develop the system?
2. What documents are produced during development?

The best answers to these questions have been debated by software practitioners for decades. Our answers are fairly mainstream, while emphasizing the unique requirements of event based programming. Our process is a simplified *waterfall model*, where programmers analyze, design, implement, test and debug the system.

- **Analysis**
  Analysis answers the question: What is the system supposed to do? It is the phase where developers work closely with clients trying to elicit the system requirements.
- **Design**
  The *design* specifies how the system will be built. Designs include what classes and objects will be part of the system, and how they will work together to get the job done.
- **Implementation**
  Implementation is a fancy term for programming. If the design is solid, the implementation follows naturally.
- **Testing**
  Testing verifies that the software operates as required.
- **Debugging**
  Debugging removes defects discovered during testing.

The documents we will produce are derived from the *Unified Modeling Language (UML)*. UML has emerged as the dominant meta-language for representing object-oriented systems. This book is not about object-oriented software engineering, and a complete survey of UML is well beyond its scope. Similarly, we are not overly concerned that the reader learns the nuances of UML. We concentrate on how UML can help us capture the essential ideas behind our event based systems. There are many good object-oriented software engineering texts, any of which will provide a more comprehensive discussion of UML and the views of the system it provides.

There are also good UML tools that support all the models we will discuss in this chapter. Some even contain code generators, where the models can be compiled, first into a high level language like Java or C++, and then into executable code.

3.4 Analyzing System Requirements

The first step in writing any program is knowing what it is supposed to accomplish. At this point we are not interested in how the program will accomplish its tasks. Instead, we just want to understand what those tasks are.

Use cases are an excellent way to document what an event based system does. Each use case describes an interaction between the user, the *Actor* in UML terms, and the system. They lay out in a step by step fashion the actions the Actor takes in a typical interaction and how the system responds to each. A complex system may have dozens of hundreds of use cases.

Use cases should be consistently documented. We suggest the following as a template:

Use Case Number: A unique number identifying this use case.
Name: Name of the use case.
Version: A version number, to track how the use case evolves.
Actor: Who or what carries out the use case.
Preconditions: What must be true before the use case begins.
Primary Path: A step by step description of the actions the actor takes during the use case and how the system responds. Numbering the steps helps, as it gives reference points for the alternative paths.
Alternative Paths: A description of alternative paths through the use case. Each alternative path uses a parallel numbering scheme as the primary path, allowing the reader to quickly see how the alternative path integrates with the primary path.

Our stopwatch only has two use cases: time an athlete in a one part event, and time a relay. Both are relatively short.

Use Case Number: 1
Name: Time A Single Athlete in a One Part Event
Version: 1.0
Actor: Coach
Preconditions:

1. The stopwatch is currently not being used to time a different event.

2. The stopwatch is stopped and reset to 0.0.

Primary Path

1. The instant the race begins, the coach presses the Start/Stop button. The watch begins displaying the elapsed time.

2. When the athlete crosses the finish line, the coach presses the Start/Stop button again. The final time is displayed on the stopwatch and recorded by the coach.

3. The coach resets the stopwatch in preparation for the next event.

Alternative Paths

1. `a.` If the race consists of multiple laps, the coach may press the Lap button at the end of each lap, recording the athlete's "split" times. The coach presses the Lap button a second time to bring the watch back to its running display.

2. Return to Step 2 of the Primary Path.

---

2. When the athlete crosses the finish line, the coach presses the Lap button. The final time is displayed on the stopwatch and recorded by the coach.

3. The coach resets the stopwatch in preparation for the next event by pressing the Start/Stop button followed by the Lap button.

This use case has two alternative paths. The first alternative path uses the notation 1.a to mean: insert this step between steps 1 and 2 of the primary path. Our second alternative path replaces steps 2 and 3 of the primary path. Note the lack of an `a.` in this enumeration.

Use Case Number: 2
Name: Time a Multi-part Event
Version: 1.0
Actor: Coach
Preconditions:

1. The stopwatch is currently not being used to time a different event.

2. The stopwatch is currently stopped and reset to 0.0.

Primary Path

1. The instant the race begins, the coach presses the Start/Stop button. The watch begins displaying the elapsed time.

2. At the end of each portion of the race, the coach presses the Lap button. The display is frozen and the coach records the elapsed time.

3. After recording the time, the coach presses the lap button again. The display is unfrozen and again displays the current elapsed time.

4. When the final athlete crosses the finish line, the coach presses the Start/Stop button. The final time is displayed on the stopwatch and recorded by the coach.

5. The coach resets the stopwatch in preparation for the next event by pressing the Lap button.

Alternative Path

4. When the final athlete crosses the finish line, the coach presses the Lap button. The final time is displayed on the stopwatch and recorded by the coach.

5. The coach resets the stopwatch in preparation for the next event by pressing the Start/Stop button followed by the Lap button.

Use cases have many strengths. They are a great tool to help elicit the *functional requirements* of the system from a client. To this end, a use case should not be full of technical jargon, but should be written at a level that both the client and the developer can understand.

Use cases are particularly useful in developing event based systems, because they often explicitly mention the sequence of input events, or, if they don't, you should be easily able to infer the sequence.

Use cases are helpful in designing user interfaces, because, if there are multiple events needed to complete the task, the user interface can be designed so that the user naturally flows through the events in sequence.

Students frequently question the amount of detail that should be included in a use case. As a rule of thumb, the first draft of a use case should try to capture the main flow of the interaction, ignoring nonessential details and alternative paths. The second draft should extend the first by including the details of the main path, and incorporating alternative paths, including how the system reacts to exceptional circumstances. In the real world, use cases are often written and rewritten many times, as clients provide more details.

A use case should be implementation independent. For example, the use case should not mention the development language or the classes or objects that will be present in the system.

3.5 Design

System design is the process of deciding how to build a system that satisfies the use cases. As you probably learned in your first programming course, there is no single correct design for any system. That doesn't mean that design isn't important, however. It is still very important that we think through our design before we start coding. A well thought out design, where each class and object have well defined responsibilities, w ill make coding, debugging and maintenance much easier.

The Model-View-Controller approach discussed in Chapter 2 provides some valuable insight into how to design an event based system.

3.5.1 Model Design

The model portion of the system consists of the data structures and other objects that make up the data that the system is storing and manipulating. The model is frequently updated by an event firing, but the model should be designed using standard object-oriented or procedural design techniques. The model should be designed in a way that is independent of whether the user interface or other system components are event based. Declare the data as private and use getters and setters to manipulate it.

The Coherence Problem

A classic problem in computer science is the *coherence problem*. The coherence problem says in a nut shell, that if a system has two or more representations of the same data, all copies must reflect the current value. The coherence problem is typically introduced to students of computer architecture when studying cache memory. Each processor core may have its own cache memory. If the same variable is stored in multiple caches, the system needs to worry about keeping them coherent.

The coherence problem occurs in GUI programming. We need to guarantee that the view reflects the current data in the model. From the design point of view, the basic problem is how to automate the model notifying other parts of the system when the its data has changed. Since this is a text on event based programming, the obvious answer is to have the model fire data changed events. A few extra lines of code in the 'setter' methods is all that is required. View components (or other objects) can register their interest in these events and update themselves appropriately[2].

The Java API contains several event classes particularly for this purpose. `ListDataEvent`s are fired by a `JList`'s model to notify the `JList`'s view that it needs to update itself. The `PropertyChangeEvent` is a more generic class for programmers to use. Java's `PropertyChangeSupport` class makes it relatively painless to add and remove `PropertyChangeListener`s. Firing a `PropertyChangeEvent` typically only requires adding a few lines of code to the setter for the property.

The Stopwatch Model

The model for our simulated stopwatch contains only two integers, one to keep track of the current elapsed time and the other to hold the time being displayed. The need for two variables may seem confusing. However, sometimes the stopwatch's display is frozen, but the elapsed time continues to update. The application needs to keep track of both.

[2]Many introductory CS students question the need for 'getter' and 'setter' methods. They favor making variables public and accessing them directly. Firing data changed events from setters is a very good example of why setters are important. The setters fire the events, but the event code remains completely invisible to the caller of the setter!

3.5.2 GUI Design

GUI design is a complex subject involving technical, psychological and aesthetic considerations. A complete discussion of GUI design is beyond the scope of this book. We adopt a simple philosophy: GUIs should be user centered and designing GUIs should be guided by common sense. Here are some basic rules of thumb:

- Make your interface clear and consistent. Think about what information the user should see and display it in a logical, easy to read way. For example, a street address should be laid out with the name on the first line, followed by the street address, followed by city, state and zip.
- Use visual cues to help the user find important information. For example, if a textfield is not editable, gray out its background. Similarly, if an operation is expected to take a significant amount of time, use a progress bar to let the user know the operation is still ongoing.
- A use case frequently contains a sequence of user initiated events. The sequence should flow naturally from the interface. For example, if three textfields need to be filled before clicking "OK", set the tab order so that the user can tab from one field to the next, and only activate the "OK" button after all three have been filled.
- Error messages should be written in language meaningful to the user. An error message that says, "You are required to enter an address" is much more meaningful one that says, "Error! Textfield addressText contains a null String". Error messages should also disappear after the error is corrected.

The Stopwatch GUI

We already saw the GUI for our simulated stopwatch in Figure 3.1. While the interface is very simple, it is worth noting that it did require some important design decisions. For example, some real stopwatches have three buttons, the third labeled "Reset". The third button separates the reset responsibilities from the lap timing responsibilities. Similarly, many stopwatches have analog displays, where several hands sweep around a clock-like face. Finally, some modern watches measure time more precisely than tenths of seconds.

3.5.3 Control Design

Control is the part of the system that determines how it will respond to events. Obviously our event handlers are a part of the control. Control is more complex than just the handlers, however. The way a system responds to events often changes during the run of the system in response to previous events. For example, most editors have an insert mode and a type over mode. Pressing the `Insert` key (an event) toggles us between the two modes. In event based programming parlance, we say that this type of program is *state based*. The events the system responds to, and how it responds, are determined by its state.

Stopwatch States

If the stopwatch is stopped, pressing the Start/Stop button starts it. If it is running, pressing the Start/Stop button stops it. The stopwatch's state determines how it responds to the event. Before designing our event handlers, our first step in designing control is to model the control states of the system.

The stopwatch is in one of four control states. They are:

- Stopped – This is the initial state. The watch is not running. Note that the state does not tell us anything about the time. The current time and display time may have been reset to 0.0, or they still may contain the values from a previous run.
- Started – The watch is running and displaying the current time.
- Lap and Started – The watch is running, but the display is frozen.
- Lap and Stopped – The watch is stopped, but still displaying a frozen time.

UML contains *state diagrams* to help developers visualize and document the control states and transition events within a system. State diagrams show the various control states of the system and the events that carry the system from one state to another. Our stopwatch only needs a basic state diagram. UML's state diagrams contain many more features than shown here.

Figure 3.2 shows a state diagram for the stopwatch. The states are represented by circles. Arrows represent transitions between states. The watch is always in one of its four states. The dark circle in the upper left is called a start marker, telling us that the stopwatch is initially in the Stopped state. From the Stopped state, the user may press the Start/Stop button which moves the watch to the Started state. To return to the Stopped state, the user presses the Start/Stop button again. These two events are represented by the two horizontal arrows labeled "Start/Stop" in the top center of the figure.

Alternatively, while in the Stopped state, the user may press the Lap button, represented by the curved arrow in the upper left of the figure. Clicking the Lap button resets the stopwatch, while leaving it in the Stopped state. Each arrow is annotated with the name of the event, followed by any action that should take place when the event occurs. In our example, "Lap" is the name of the event and appears to the left of the /. To the right of the / is "Reset Time and Display", the action to take place.

After the stopwatch is started, it receives ClockTick events. ClockTick events are received in both the Started state and the Lap-And-Started state. These events are generated by an internal timer. They are responsible for updating the stopwatch's internal time and the displayed time. In the Lap-And-Started state, only the time is updated; the display is frozen.

The lower two states in the figure represent the stopwatch when the display is frozen. To freeze the display, the user presses the Lap button while the stopwatch is running (in the Started state). The stopwatch transitions to the Lap-And-Started state. It remains running, but the display is frozen. ClockTick events are processed behind the scenes, updating the elapsed time, but not the display time. The user has no visual evidence that they occur. The Lap-And-Started state appears in the lower right of the figure. From Lap-And-Started the user may return to the Started state by again pressing the Lap button.

From Lap-And-Started the user may also press the Start/Stop button. The transition is to the Lap-And-Stopped state. This transition has no visible effect. The displayed time was already frozen and it remains so. The change is behind the scenes. In Lap-And-Stopped the watch is not running and ClockTick events are ignored. The user may return the watch to the Lap–And–Started state by again pressing the Start/Stop button.

From the Lap-And-Stopped state, the user may return the watch to the Stopped state by pressing the Lap button. This event causes the display time to be updated to the final elapsed time.

Implementing Control State

There are several alternatives for implementing control state.

We could use an integer variable to store the control state.

- `state == 0` means the watch is in the Stopped state.

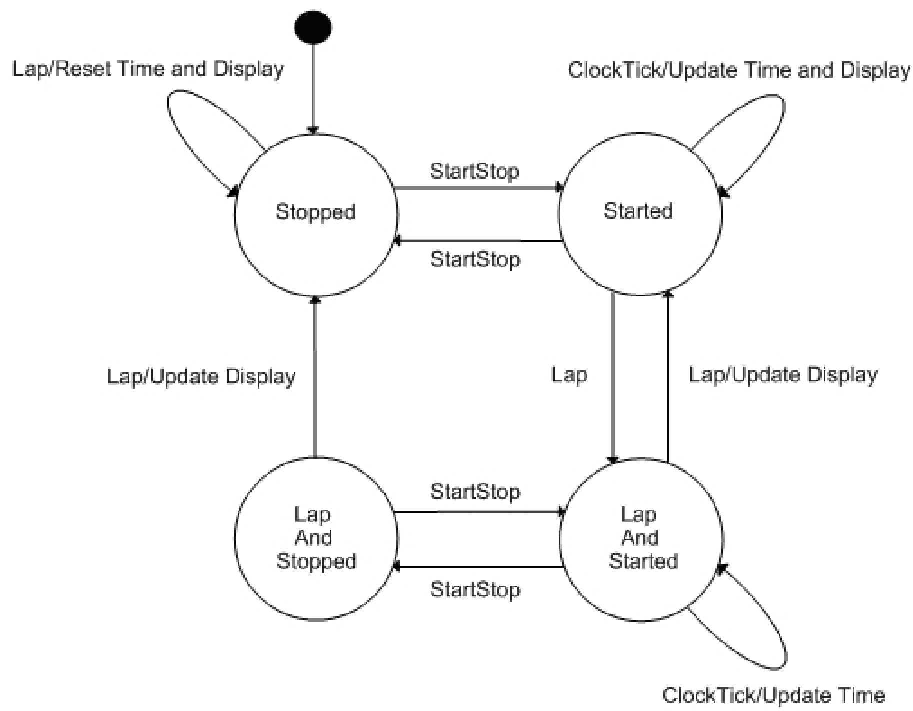

Figure 3.2: The state diagram for the two button stopwatch.

- `state == 1` means the watch is in the Started state.
- etc.

In this system, each event handler would contain a case statement based on the control variable and carry out different actions accordingly. This approach works. Its main problem is with it is repeated program logic. If our state machine design contains a bug, say we need a fifth state, then every one of the case statements would have to be modified. In a system with hundreds of events, this becomes a maintenance nightmare.

Another approach would be to de-register our handlers and re-register new handlers for each new state. We have eliminated the case statements, but there are more handlers. Control state in this approach is represented by the set of currently registered handlers. Like the previous approach, this approach works. Unfortunately, it suffers from the lack of an explicit representation for the control state. For example, if there is a runtime error, we can't easily determine the control state when it occurred. We have to look at all the registered handlers, a nontrivial task.

The best approach is to use the *State Design Pattern* [Gamma et al., 2000]. In the state design pattern, control state is modeled using objects. A `State` interface defines the methods each state class must implement. There is a different class and corresponding object for each control state. Each state contains a method for each possible event. The program contains a variable of type `State`, named `state`, storing the system's current state.

For example, the `State` interface declares `public void startstopPushed()`. The startstop handler will call `state.startstopPushed()`. The actual method called will vary based on the `state`, but that doesn't matter to the handler.

The event handlers delegate responsibility for the event handling to the `state` object. Using polymorphism, they call the `state` object's methods. The handlers don't know, nor need to know, what the control state is. They just know that the `state` variable is there and contains the needed methods.

Figure 3.3 shows the structure of the State Design Pattern.

A major advantage of the State Design Pattern is that event handlers are now independent of control state and the control states are independent of each other. If more states are added, more concrete state classes are implemented, but the handler code and the code in other states remains unchanged.

State Classes for the Stopwatch

Figure 3.4 shows the state classes for the Stopwatch.

The state interface for the Stopwatch contains three method declarations:

- `updateTime()` – which responds to the ClockTick event,
- `startStopPushed()` – which implements the activities to perform when a startStop event occurs, and
- `lapPushed()` – which implements the activities to perform when the lap button is pushed.

Each concrete state class implements each of these methods, thereby supplying the appropriate action for each event.

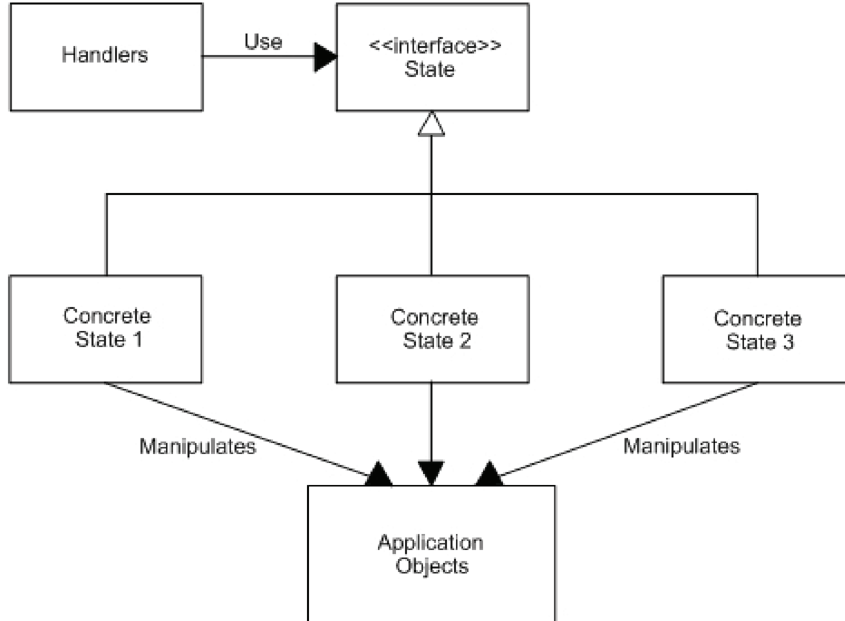

Figure 3.3: The *State Design Pattern* suggests using a different class for each control state present in the system. Handlers invoke methods in the `state` object to manipulate application objects. The handlers do not need to concern themselves with what the control state is, only that it exists.

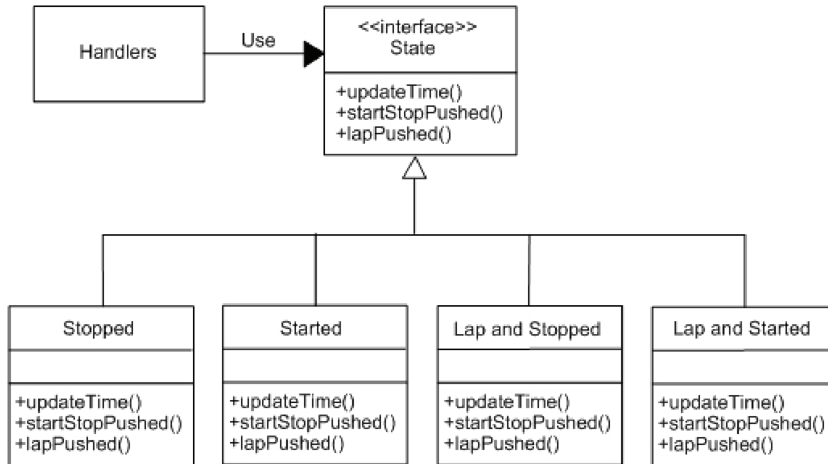

Figure 3.4: The control states for the stopwatch.

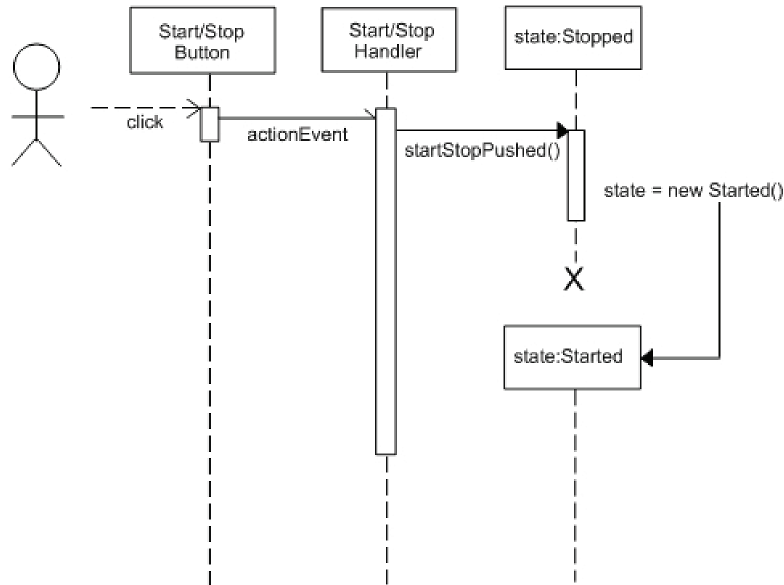

Figure 3.5: This sequence diagram shows the events and method calls made when the Start/Stop button is pushed while the stopwatch is in the Stopped state.

Designing Event Handlers

Designing and implementing handlers is much simpler if we have done a good job designing our data model and our control state. Regardless of the type of system being constructed, event handlers have three distinct responsibilities:

- Update the program's data.
- Update the control state.
- Notify other objects that the event has occurred.

In simple programs, with only one control state, the handler can take care of these itself. If the system is state based, the responsibilities are delegated to the `state` object.

Sequence diagrams are included in the UML to show the sequence of events and method calls during an interaction. Objects and classes appear as boxes along the tops of the columns. The lifetime (extent) of an object is represented by a dashed vertical line extending downward from it. Method invocations are shown by elongated rectangles along the object's life line. Arrows show events, method calls and return values.

Figure 3.5 shows a sequence diagram for pressing the Start/Stop button while the stopwatch is stopped. The user clicks the button which fires an `ActionEvent`. The event is handled by the Start/Stop handler, which in turn delegates the details of the event handling to the state. In this case, the state replaces itself with a new `Started` object.

During this event, there are no program data to update, nor does any other object need to be notified. If these actions were required, two additional arrows would emerge from the right side of the state object's method call.

Exception Handling in Event Based Programming

Exceptions occur when something unexpected happens during the run of a program. Some exceptions are due to poor programming, like running off the end of an array. Other exceptions are caused by unexpected user input, like typing alphabetic characters when a number is expected. In some ways exceptions are like events. They occur at a specific point in time. They should be handled to prevent the program from crashing.

The two primary differences between exceptions and events are that:

- Exceptions are handled within the thread of execution where they occur. That is, if method A calls method B and method B calls method C and an exception occurs in C, it should be handled in C, B, or A, not elsewhere. This is different from event handling, where the event handlers generally execute in a separate thread of execution from the event source.

- Exceptions should be handled immediately. They represent a problem with the program and that problem needs to be fixed before the program continues execution. Events, by contrast, are a normally occurring part of the program's execution and are handled asynchronously in many infrastructures.

One of the practical questions when dealing with exceptions is where they should be handled. For example, Java gives multiple syntactic structures,

- `try .. catch`
  handles the exception locally, method C in the above example, and

- `throws`
  handles the exception further up the call stack, methods B or A in the above example.

There is a clear answer to this question in event based programming: *Event handlers should handle all exceptions that may occur during their execution.* They should not try to throw the exception elsewhere. Event handlers have a purpose. They know what they are trying to accomplish. They know what the exception is. They have access to all the objects that might need modifying to correct the exception. They should take care of the problem.

As discussed above, event handlers manipulate the model and the control state. Exceptions may be thrown in by the model or control. They should be caught by the event handler, however.

This approach leads to resilient programs that keep on working even after problems occur.

3.6 Stopwatch Implementation

Our stopwatch design may be implemented directly into Java, C# or any other object–oriented language. In this section we present snippets of the Java code. Entire Java and C# implementations may be found on the book's website.

3.6.1 The Java Source Code

In this section we highlight a few of the interesting aspects of the Java implementation of the stopwatch.

Coding control states in Java requires an interface with a concrete implementation of it for each control state.

```
93    // State is the interface that all concrete states must implement
94    public interface State {
95       public void updateTime();        // run at a timer event
96       public void startStopPushed(); // run when startStop is clicked
97       public void lapPushed();         // run when lap is clicked
98    }
```

The interface contains a method for each of the possible events.

```
126   // The state when the stopwatch is running
127   public class Started implements State {
128      public void updateTime () {
129         setCurrentTime(getCurrentTime()+1);
130         setDisplayTime(getCurrentTime());
131      }
132      public void startStopPushed()
133         state = new Stopped();
134      }
135      public void lapPushed() {
136         state = new LapAndStarted();
137      }
138   }
```

Java source code for the Started state appears above. Each method body is completed with the actions to take for that event. `updateTime()` is called when the the timer event fires. Because the watch is running and displaying in this state, it updates both the `currentTime` and the `displayTime`. The other two events modify the control state. They are direct translations of two of the transitions in Figure 3.2.

```
161   // The LapController simply dispatches the event to the state
162   public class lapController extends AbstractAction {
163      public void actionPerformed (ActionEvent e) {
164         state.lapPushed();
165      }
166   }
```

The handler for the lap button is shown above. All handlers become one liners, delegating full responsibility for handling the event to the control state.

3.7 Testing

Before discussing event based testing, we remind you that many of the systems we are discussing are not totally event based. They often contain object-oriented (or procedural, or functional) code, as well. This code should be tested and debugged using appropriate techniques and tools for that paradigm.

Documenting and implementing a test plan should be done in a disciplined way. Unfortunately, introductory CS students often think that testing a program means running it against the sample data on the assignment sheet. If the sample output is produced, the program is assumed correct. Testing takes more work than that. We should plan what tests we intend to run and know what results are expected. Then, when the tests are run, we should document what the actual results were and whether the test passed. We suggest a test form similar to the following:

Test Number: The number of this test.
Test Name: Descriptive name of the test.
Preconditions: List must be true before the test is run. Note that there might be considerable setup code to get the system into the correct initial state, before carrying out the test.
Expected Results: List the output and changes to the system that are expected as a result of the test run.

Then for each time the test is run:
Date and time:
Date and time the test was carried out. **Results Obtained:** The actual results
Pass/Fail: Whether the system passed the test.

Testing and debugging event based systems pose unique challenges in part because many of the advanced language features they adopt, like multithreading, inheritance and polymorphism. The added layers of complexity make it both easier to introduce bugs and harder to find them. The event based programming community is in the early stages of developing methodologies to cope with these complexities [Beer and Heindl, 2007], but the authors know of no well defined comprehensive approach to testing event based systems. The Extreme Programming community claims that tests should be designed to test what can go wrong [Beck and Andres, 1999]. This strikes us as good common sense advice, and our discussion follows that philosophy.

As we have seen, the event based portion of a project consists of event sources, event objects, and event handling code – both event handlers and control state. When developing GUI programs, the event classes and source classes are often supplied by an API and don't require further testing. Therefore, our approach emphasizes testing the event handling code.

3.7.1 Testing Event Handling Code

Event handlers are the glue that holds our application together. They are typically tightly coupled to many different objects. They frequently affect both data state and control state, and they may fire more events themselves. Testing event handlers in isolation is very difficult, requiring an extensive test harness. Another practical problem arises because of the common use of anonymous inner classes to implement handlers. It is not possible to instantiate an anonymous inner classes to test it separate from their surrounding class.

We recommend treating event handler testing as a type of integration testing. That is, we test fire events within the context that they will actually run, not with stubbed in code. We suggest two suites of tests; a suite oriented around control state, and a suite oriented around use cases.

State Based Testing

Each event should be test fired in each control state. In our stopwatch there are four control states and three events (two button clicks and a timer event). This gives us a total of 12 tests. For example, when the stopwatch is running, does pushing the Start/Stop button produce the correct result? Does pushing the Lap button produce the correct result? Does the timer firing produce the correct result? We then repeat the same three events with the watch in the Stopped and in the two lap states, as well.

When testing any code, it is important to define what it means to pass a test. What does the phrase used above, "produce the correct result" mean? When we discussed design earlier in this

chapter, we said that event handlers take three different types of actions. They update data state, update control state, and notify others that the event has occurred, possibly by firing further events. The expected result of each test firing is the updated values of data state and control state, and whether the notifications took place.

Use Case Based Testing

The second suite of tests we recommend are based on the use cases. A test should be designed for each path through the use case. In our stopwatch example there are two use cases. This suite of tests should include:

1. Test timing a one part event.
2. Test timing a one part event with multiple laps.
3. Test timing a one part event where the lap button is used at the end of the event.
4. Test timing a multi-part event.
5. Test timing a multi-part event where the lap button is used at the end of the event.

Even though we have previously tested each event in each control state, bugs are still frequently detected during use case testing that are not caught earlier.

3.7.2 Testing Event Sources

We want to briefly discuss testing systems where we have developed our own types of events,including event sources. The event source classes should be put through at least the following tests:

1. Event sources should be tested to see that event handlers are correctly registered, including registering multiple handlers, if appropriate.
2. Event sources should be tested to see that event handlers may be de-registered, including removing all handlers.
3. Each method that fires events should be tested with zero, one and multiple handlers to make certain that all handlers are notified.

3.8 Debugging

Not so surprisingly, students (and instructors and professional programmers) introduce the same types of errors in event based programs over and over. While good debugging skills need to be developed over time with lots of patience and practice, knowing some of the places and reasons bugs are introduced can be a big help.

3.8.1 Handler Registration

The dynamic registration of event handlers with event sources is similar to polymorphism in object-oriented programs. It differs markedly, however, in several respects. The mapping between sources and handlers is not one to one. A single event source may have multiple handlers associated with it (*multicasting*). Similarly, it is possible for a handler to be listening for events from several sources (*multiplexing*). Each registration of a source with a handler is a unique computation. Finding that several of them have been established does not guarantee that all of them have. Our best advice

here is to avoid de-registering and re-registering handlers. Register the handlers once, early in the life of the system, and leave them registered. Changes in control state can be handled using the state design pattern rather than changing handlers. This solution is much cleaner and will result in fewer bugs.

3.8.2 Nondeterminism

Another difference between event handling and polymorphism is that many systems use dedicated event handling threads. The event source's thread does not wait for the handlers to complete before generating more events or taking other actions. This makes it impossible to know if event firings and event processing will happen identically curing two apparently identical runs of the same test code. This means that a test may pass one time and fail the next, even though no code was modified.

3.8.3 Cascading Events

Event propagation poses still more issues. As we have seen in some of our examples, firing an event can cause long chains of additional events. In fact, if our events are multicast, we can have an entire web (or spaghetti bowl) of events propagating changes throughout our code. The difficulty is one of the sheer complexity of the situation. Did all of the appropriate events fire? What were all the appropriate changes to the data state and control state? Did all the values change appropriately? If not, where was the chain broken? The event based programming paradigm is currently at the state that procedural programming was in the 1960s, before the introduction of the disciplined use of structured programming. Goto statements abounded then. Now cascading events may abound.

3.9 Refactoring the Stopwatch

Our two button stopwatch example has served us well. The implementation reflects the way stopwatches actually work. However, there are several design flaws that should be recognized. The first flaw is that the Lap button is overworked. It freezes and unfreezes the display and, when in the stopped state, it resets the stopwatch. The semantics of the stopwatch are simple enough that most users have no trouble understanding it, but still, it is not a good design decision to have the same event source filling two very different roles. In a more complex system overworking input components and events is undesirable. Some stopwatch manufacturers recognize this and produce three button stopwatches, with separate Lap and Reset buttons.

Another problem with the stopwatch design is that the state semantics are tightly coupled. There are really two different aspects of control, starting and stopping the stopwatch, and freezing and unfreezing the display. In the current implementation, the Lap button plays a role in both aspects. This coupling creates maintenance problems. Adding, deleting or modifying any state will probably require changes in the other states. There are only four states in the stopwatch, so this tight coupling does not pose major problems, but as a general design consideration partitioning the states into disjoint subsets for controlling model and view provides a more understandable and maintainable design.

In this section we refactor our stopwatch design and implementation to address both of these problems.

3.9.1 Control and View Control

In our modified stopwatch design, we separate control of the display from start/stop control. Starting and stopping the watch is handled as before. Freezing and Unfreezing the display is now a separate

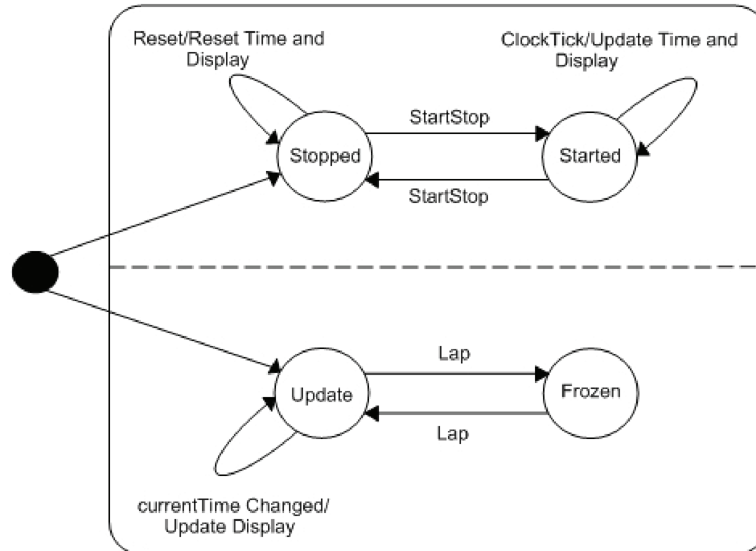

Figure 3.6: The state diagram separates Control and View portions of the stopwatch. Control contains two substates, Stopped and Started. The view also contains two substates, Update and Frozen.

state machine.

In addition, we further decouple views of the stopwatch from the model by making them event based. View(s) will receive stopwatch events when the current time is updated. The stopwatch fires events to notify the view(s). This makes it easy to have multiple views of the same watch. UML's state diagrams represent concurrent state machines by separating them with a dashed line. Figure 3.6 shows the updated state diagram.

In the new design, the controllers remain tightly coupled to the model. The controllers access the model via its methods. In Java GUI programs, control and model generally remain tightly coupled. The controller directly calls the model's methods to update the model state. This coupling doesn't pose a problem, as long as the controllers use accessor methods rather than direct manipulation of the model's data.

3.9.2 The Modified Java Source Code

The Java source code is modified from the two button stopwatch as follows:

- All attributes related to the display, including the display time, the textfield for the display and the lap button are removed from the stopwatch class and and placed in a separate View class.

- A `Reset` button is added to the stopwatch class.

- `PropertyChangeEvent`s notify the views that they need to be updated.

The code fragments below show the major modifications to the stopwatch source code. The entire source code for the refactored stopwatch is available on the text's website.

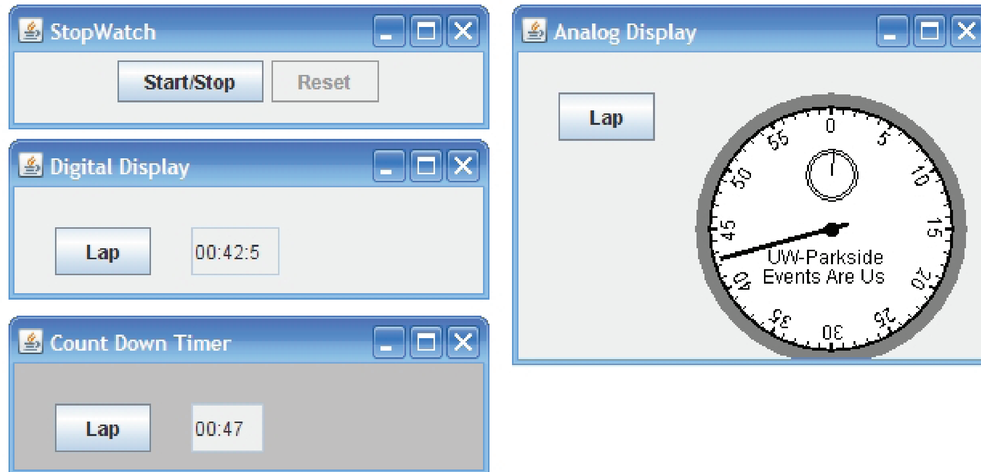

Figure 3.7: The control for the stopwatch is shown in the frame with the Start/Stop and Reset buttons. Each of the other frames shows a distinct view of the stopwatch. The Analog and Digital displays show the same time, represented in very different formats. The countdown displays the amount of time remaining from an initial value of 90 seconds.

```
13    private long currentTime;      // the current time
14
15    private Timer timer;           // the timer goes off every 1/10 seconds (or so)
16    private State state;           // the running state of the stopwatch
17    private JButton startStop;     // the start stop button
18    private JButton reset;         // reset the stopwatch to 0
```

Lines 13-18 declare the variables for the model and controller. There is no longer a lap button or a display textbox. Instead, we find the reset button.

```
66    // When we set the time, we fire off an event
67    public void setCurrentTime (long timeArg) {
68        long temp = getCurrentTime();
69        currentTime = timeArg;
70        firePropertyChange("currentTime", new Long(temp), new Long(currentTime));
71    }
```

`setCurrentTime()` is modified so that it fires a `PropertyChangeEvent` to notify the views that they should update themselves. In Java, `PropertyChangeEvents` only fire if the old and new values differ. Therefore we pass the original time along in `temp`, so that the `firePropertyChange()` method can decide if an event needs to fire.

```
89    // addView sets up a new viewer for the stopWatch
90    public void addView (SWView view) {
91        addPropertyChangeListener(view);
92        view.addPropertyChangeListener(new TimeRequestController());
93    }
```

We modify the stopwatch with an `addView()` method to register views with the stopwatch.

There is one special case that requires additional comment. If the stopwatch has been stopped after the view is frozen, the current time is later than the display time. When the view is unfrozen, we need some mechanism for the display time to be updated. The stopwatch is stopped and the current time isn't changing, so the model is not firing events. The `TimeRequestController` is our way to handle this situation. The view fires an event to the `TimeRequestController`, requesting it to send the current time, which it does.

An interesting alternative to this approach is to have the `Stopped` state continue to fire time update events, even though the stopwatch is not running. The watch will fire events containing the same time over and over. Now, when a view is unfrozen, it will receive the correct time without doing any additional work.

```
155
156     // This class listens for requests for the current time
157     public class TimeRequestController implements PropertyChangeListener {
158         public void propertyChange (PropertyChangeEvent evt) {
159             firePropertyChange("currentTime", null, new Long(currentTime));
160         }
161     }
```

The `TimeRequestController` is an inner class to the stopwatch. It fires a property change event notifying all views of the current time. The second parameter of the `firePropertyChange()` is set to null, so the comparison with the current time will always fail and the event will fire.

Multiple Views

It is now easy to implement multiple views of the stopwatch. The only way the views differ is in how they display the stopwatch's time, so most of the work for a view can be done in an abstract class. Each view contains two states, Updating and Frozen, and a Lap button to toggle between them. The code for the three different views of the modified stopwatch can be found in the appendices and on-line. For brevity, it is not shown here.

3.10 Summary

In this chapter we have presented a relatively simple example of an event based system. It illustrated many basic features of event based systems implemented in object-oriented languages. Model-View-Controller and state design patterns were shown to be applicable. Several different UML models were used, including class diagrams, sequence diagrams and state diagrams. The state diagrams served as the basis for designing and implementing the different versions of the stopwatch.