Event Based Programming

Article

2 authors, including:

Timothy V. Fossum
State University of New York at Potsdam
42 PUBLICATIONS **343** CITATIONS

Event Based Programming

S. Hansen
UW - Parkside

T.V. Fossum
SUNY - Potsdam

Kenosha WI, May 23, 2010

Contents

1 Event Based Systems **1**
- 1.1 Events . . . 1
  - 1.1.1 Responding to events . . . 1
  - 1.1.2 Event sources . . . 2
- 1.2 Event based systems . . . 2
  - 1.2.1 Request-response . . . 2
  - 1.2.2 Message passing . . . 2
  - 1.2.3 Publish-subscribe . . . 3
  - 1.2.4 Events in Systems . . . 3
  - 1.2.5 System state . . . 3
  - 1.2.6 Event based systems defined . . . 4
  - 1.2.7 Discrete systems and events . . . 4
  - 1.2.8 Examples of events . . . 5
- 1.3 Attributes of Event Based Systems . . . 6
  - 1.3.1 State Based . . . 7
  - 1.3.2 Nondeterminism . . . 7
  - 1.3.3 Loose Coupling . . . 8
  - 1.3.4 Decentralized Control . . . 8
- 1.4 The Event Based Programming Paradigm . . . 8
  - 1.4.1 The Event Model . . . 8
  - 1.4.2 Events versus Method Invocations . . . 9
  - 1.4.3 Writing Event Based Programs . . . 10
- 1.5 Goals of the Text . . . 10

2 Event Based Programming Basics **13**
- 2.1 Introduction . . . 13
  - 2.1.1 Integrated Development Environments . . . 13
- 2.2 The Object–Oriented Event Model . . . 14
  - 2.2.1 A Simple Example . . . 14
- 2.3 Java Language Features to Support Event Based Programming . . . 17
  - 2.3.1 Anonymous Classes . . . 17
  - 2.3.2 Inner Classes . . . 19
  - 2.3.3 List Example using Inner Classes . . . 19
- 2.4 Inheritance . . . 23
  - 2.4.1 Model–View–Controller . . . 23
  - 2.4.2 The Drawing Application . . . 33
- 2.5 List of Doubles . . . 36
  - 2.5.1 Double List Model . . . 38

2.5.2 Average and Max Classes . . . 40
2.5.3 The Main Class . . . 42
2.5.4 Event Handler Classes . . . 45
2.5.5 Event Propagation . . . 48
2.6 Procedural Event Programming . . . 48
2.7 Summary . . . 49

3 Software Engineering Event Based Systems 51
3.1 Introduction . . . 51
3.2 The Two Button Stopwatch . . . 52
3.3 Event Based Software Development . . . 53
3.4 Analyzing System Requirements . . . 54
3.5 Design . . . 57
3.5.1 Model Design . . . 57
3.5.2 GUI Design . . . 58
3.5.3 Control Design . . . 58
3.6 Stopwatch Implementation . . . 64
3.6.1 The Java Source Code . . . 64
3.7 Testing . . . 65
3.7.1 Testing Event Handling Code . . . 66
3.7.2 Testing Event Sources . . . 67
3.8 Debugging . . . 67
3.8.1 Handler Registration . . . 67
3.8.2 Nondeterminism . . . 68
3.8.3 Cascading Events . . . 68
3.9 Refactoring the Stopwatch . . . 68
3.9.1 Control and View Control . . . 68
3.9.2 The Modified Java Source Code . . . 69
3.10 Summary . . . 71

4 Event Infrastructure 73
4.1 Introduction . . . 73
4.2 The Event Life Cycle . . . 74
4.2.1 Event Handling Requirements . . . 75
4.2.2 Timing Requirements . . . 75
4.2.3 Concurrent Event Handling . . . 75
4.3 Event Infrastructure Services . . . 76
4.3.1 Event Registration . . . 76
4.3.2 Event Dispatching . . . 76
4.3.3 Hybrid Dispatching . . . 80
4.4 Java's Support for Events . . . 80
4.4.1 Handler Registration . . . 80
4.4.2 JCL's Event Classes . . . 81
4.4.3 Java Event Processing . . . 81
4.5 Programmer Defined AWT Events . . . 85
4.5.1 New Event Classes and Interfaces . . . 87
4.5.2 The `BloodPressureListener` Interface . . . 87
4.5.3 The `BloodPressureEvent` class . . . 88
4.5.4 The `Patient` class . . . 90
4.6 The Beans-Like Implementation . . . 91

4.6.1 Revamped Blood Pressure Events . . . 91
4.6.2 The Revamped Patient Class . . . 92
4.6.3 Infrastructure or Client Code . . . 93
4.7 Summary . . . 93

5 Threads and Events **95**
5.1 Introduction . . . 95
5.2 Background . . . 95
5.2.1 Threads and Concurrency . . . 96
5.3 Why use Threads . . . 96
5.4 A Multi-Threaded Event Handler . . . 99
5.4.1 Simple Handler . . . 100
5.4.2 Threaded Handler . . . 100
5.5 Problems Arising from Multi-Threading . . . 101
5.5.1 Problems with Resource Sharing . . . 101
5.6 Threads as an Event Based System . . . 102
5.6.1 State Based . . . 103
5.6.2 Nondeterminism . . . 104
5.6.3 Loose Coupling . . . 104
5.6.4 Decentralized Control . . . 105
5.7 Summary . . . 105

6 Distributed Event Programming **107**
6.1 Introduction . . . 107
6.2 The Big Picture . . . 107
6.3 Distributed System Infrastructures . . . 108
6.4 Event Based and Distributed Systems . . . 110
6.5 Publish - Subscribe . . . 111
6.6 Challenges Developing Distributed Systems . . . 111
6.7 The CORBA Infrastructure . . . 112
6.7.1 Services, Tools and Classes . . . 112
6.7.2 CORBA Programming Steps . . . 115
6.8 Calculator Example . . . 115
6.8.1 Defining the IDL . . . 116
6.8.2 Compile the IDL file . . . 116
6.8.3 Completing the Server . . . 118
6.8.4 The Client . . . 121
6.8.5 Starting the Application . . . 124
6.9 Asynchronous Method Invocation . . . 124
6.9.1 CORBA and Asynchronous Messages . . . 124
6.9.2 Heart Monitor Example . . . 125
6.10 Peer to Peer Distributed Applications . . . 126
6.10.1 CORBA Peer to Peer Applications . . . 126
6.10.2 Chat Example . . . 126
6.11 Conclusion . . . 130

7 Events and the Web **131**
7.1 Introduction . . . 131
7.1.1 Historical Perspective . . . 132
7.1.2 Multi-tiered Architectures . . . 132
7.2 Web Fundamentals . . . 132
7.2.1 HyperText Transfer Protocol (HTTP) . . . 133
7.2.2 HyperText Markup Language (HTML) . . . 133
7.2.3 Web and Application Servers . . . 135
7.3 Java Servlets . . . 135
7.3.1 Calculator Example . . . 135
7.4 Adding State to Web Applications . . . 141
7.4.1 Cookies . . . 141
7.4.2 Sessions . . . 141
7.4.3 Other Ways to Maintain State . . . 145
7.4.4 Combining Request and Response Pages . . . 145
7.5 Java Server Pages (JSPs) . . . 147
7.6 Web Services . . . 149
7.6.1 Stock Quote Example . . . 150
7.6.2 The eXtensible Markup Language (XML) . . . 150
7.6.3 Finding a Web Service and its API . . . 151
7.6.4 Summary of XML Uses in Web Services . . . 152
7.7 Developing a Web Service . . . 153
7.7.1 Quadratic Equations Revisited . . . 153
7.7.2 Initial Steps . . . 154
7.7.3 Web Service Annotations . . . 154
7.7.4 Completing the Web Service . . . 155
7.7.5 Testing and Deploying the Web Service . . . 156
7.7.6 WSDL and Schema Revisited . . . 156
7.8 Developing a Web Service Client . . . 158
7.8.1 Initial Steps . . . 158
7.8.2 Completing the Client . . . 158
7.9 Making Web Services More Event Based . . . 159
7.9.1 Developing a Web Service with State . . . 160
7.9.2 Client for a Web Service with State . . . 163
7.10 The Changing Landscape of Web Services . . . 164
7.10.1 Web Services Inspection Language(WSIL) . . . 164
7.10.2 SOAP versus JSON . . . 164
7.10.3 RESTful Web Services . . . 165
7.10.4 Evolving Language Tools . . . 165
7.11 Conclusion . . . 166

Preface

Introduction

This is a book about event based programming. There are dozens of computer science books that have 'event' or 'event based' in their titles. Almost all of these are about some particular language or system that uses events. This book is different. Our goal is to introduce you to event based programming – and, more generally, event based systems – as a computer science paradigm that focuses on the fundamental ideas relating to understanding, designing, implementing, and testing loosely coupled systems.

One of our objectives is to introduce you to approaches used in event based programming and to illustrate them with carefully crafted examples. Programming event based systems is different from procedural programming or object-oriented programming. If the solution is to be event based, programmers think about the problem differently. Its important that you understand why an event based approach is more appropriate for some problems than for others.

Event based programming also has its own set of challenges, principally because almost all event based systems are nondeterministic and are inherently difficult to test and debug. Another of our objectives is to show you how to use conceptual and design tools to create event based systems that behave correctly.

What you will learn from this book is applicable to any event based language or library. Since this is a book about programming, you will encounter many programs as you read it. We have chosen Java as our principal implementation language. Java is available almost universally for hardware platforms and operating systems, and Java has native support for events. However, this book is not about making you an expert in Java. Instead, you will gain an understanding of how to develop event based software, with Java serving as a particular implementation language. In the end, you will be able to develop reasonably sized event based applications in Java, and you will be able to take the principles and ideas you learn and apply them to new and different event based languages as needed.

Computing is continually changing. Computing professionals are always playing catch-up, trying to keep their technical knowledge current with regard to new languages, operating systems, and application versions. On the other hand, programming paradigms change very little. Procedural programming – with its while loops, if statements, and procedure calls – has remained largely unchanged since the 1950s. Similarly, the concepts behind object-oriented programming have been relatively stable for over 20 years. Procedural programming and object-oriented programming are *paradigms*. They aren't about any particular language. Instead, they give the programmer problem solving techniques and methodologies that are applicable in a variety of languages.

In recent years, event based programming has emerged as its own distinct paradigm. Its notions of runtime association of sources and handlers, and minimal timing assumptions apply, and will continue to apply, no matter how many new event based languages are released. The fundamental ideas behind the paradigm are stable. By emphasizing an understanding of the paradigm, we will

lay a foundation that will let you easily pick up new event based languages as they emerge.

Events are incredibly important in modern computing. They occur in graphical user interfaces (GUIs), operating systems, discrete event simulation, database management systems and many other computing-related fields. They are also integral to the operation of user interfaces in modern electronic devices such as cellular phones and television sets. Because events are so ubiquitous, it is reasonable to expect that you, as a student of computing, should understand them in detail. Unfortunately, you are more likely to learn first about event based programming when you are asked to develop GUIs – and then you would typically be exposed only to what you need to make the GUI operate. As you will see, events have many more applications than implementing GUIs.

Event based systems are commonplace, but they have distinct properties and pose unique challenges for developers. Event based programming deserves a broad–based comprehensive treatment in the computer science curriculum. This book is an effort to provide that treatment.

Audience

You should be comfortable approaching the material in this book if you have a background equivalent to a two-semester undergraduate-level course in programming with some experience using an object-oriented language. Knowing Java is a plus, but you should be able to understand the programming examples without it.

You should be comfortable dealing with elementary data structures such as arrays and lists. We may refer occasionally to more advanced data structures, in which case we will provide background discussion and pointers to resources where you can study relevant material. You should also be acquainted with classes, objects, inheritance, and polymorphism, though again we may take the opportunity to refresh your knowledge of these ideas when they play a particularly important role in our discussions or in example code. We do not expect you to have studied mathematics beyond elementary calculus and/or discrete math.

Resources

While Java is our primary implementation language in this book, other languages (C++, Python, and C#, for example) – along with appropriate library support – can serve equally well. We will post sample code on our course website (`http://cs.uwp.edu/Events`) to supplement the code in this book as it becomes available.

Chapter 1

Event Based Systems

1.1 Events

An *event* is an observable occurrence. An "occurrence" is something that happens at some point in time. An occurrence is "observable" if it is possible for an observer to notice that it happened. (If a tree falls in the forest and nobody is there to hear it, the event still occurs since it is *possible* for an observer to have heard it.)

As you can imagine, events are happening continually. Here are some examples of events:

- a raindrop falls into a river
- a river overflows a dike
- a star goes supernova
- an insect flies nearby
- a driver presses the brake pedal of a car
- a person sends a FAX to a remote FAX machine
- a FAX machine receives a fax transmission

1.1.1 Responding to events

If an observer takes an interest in an event, the observer may respond to the event in some way. If you are a farmer, you are not likely to take an interest in a particular raindrop falling into the river next to your field, but you may well take an interest in the river overflowing a dike and flooding your field. If you are an astronomer, you may take an interest in a supernova event if you are the first to discover it, and if so you may respond by announcing your discovery in a science blog. As a human, you may not take an interest in an insect flying nearby, but a frog may well take a keen interest in such an event.

Using the jargon of programming, when an observer responds to an event, we say that the observer *handles* the event and that the observer is the event *handler*. We will use this terminology throughout the remainder of the book – though *responds* and *observer* are entirely appropriate terms to describe the same ideas in non-programming situations.

1.1.2 Event sources

Some events occur because some agent was responsible for causing them to occur. In the above examples, the driver pressed the brake pedal and the person sent the FAX; each caused the particular event to occur. Again using the jargon of programming, we say that the agent causing the event is the event *source* and that the agent *fires* the event. This terminology is also used in neurobiology where one would say that a neuron *fires*; of course, this too is an event.

When an event occurs but we are not particularly interested in the source of the event, we may say informally that the event fires – though events themselves are not agents capable of action.

1.2 Event based systems

We define a *system* to be a collection of agents subject to a set of defined behaviors and interactions. In a system, an agent's behavior at any point in time is dependent on the agent's *state*. Agents in a system can interact among themselves in several ways. We describe three event based interaction types here:

1.2.1 Request-response

A *request-response* interaction (also called *request-reply*) is between two agents. Agent A makes a request to agent B by sending agent B a request indicating the type of request along with the details of the request. Agent B processes the request and responds by sending a reply back to agent A.

Here are two examples of request-response interactions:

- Mark drops his car off at his favorite repair shop, "B-2 Automotive Repair", and requests that they fix his car's broken water pump. When the repair shop has completed the job, Mark drives his repaired car home.
- A web browser requests to load this book's homepage. The web browser waits for the server to respond with a copy of the page. T he browser proceeds to display the page on the screen.

When the requesting agent waits and does nothing until the response arrives, the interaction is called *blocking* or *synchronous*. The second example above is blocking, since the web browser waits for the server to respond. The first example above may or may not be blocking, depending on whether Mark waited at the repair shop (sleeping on the waiting room couch, perhaps) for the repair job to be finished, or he went home (because the repair shop needed to order a new pump) and did other things until the repair shop called to say that the car was fixed. If an interaction is not blocking, we say it is *nonblocking* or *asynchronous*.

1.2.2 Message passing

A *message passing* interaction is also between two agents. Agent A sends a message to agent B containing the type of the message along with any message details. Message passing differs from request-response in that once agent B receives the message, the agent is not required to respond to agent A.

Here are two examples of message passing interactions:

- "XYZ Brokers" sends a spam email to Beverly about a hot stock prospect that (they say) will double in price in the next two days.
- Mark receives a voice message from his dentist's office that Mark missed his teeth cleaning appointment.

1.2.3 Publish-subscribe

A *publish-subscribe* interaction involves multiple agents. A gents $B_1, B_2, \cdots, B_m$ subscribe to a message service indicating that they want to receive certain types of messages. Agents $A_1, A_2, \cdots, A_n$ publish various types of messages to the service. If agent A_i publishes a message type that agent B_j is interested in, agent B_j will receive a copy of the message.

Here are some examples of publish-subscribe interactions:

- Ahmed and Beverly subscribe to the "Weaving Monthly" magazine and receive copies of the magazine through the postal service as they are published.
- Mark checks a "supernova watchers" wiki on a regular basis. Sonia and Ahmed frequently post to the wiki, and Mark reads their posts.
- Pat subscribes to an Internet service that publishes real-time stock transactions, but Pat only receives transaction information from the two stocks with ticker symbols VGZ and BAA that she has subscribed to.
- An object in a Java program requests to receive all mouse click events that occur on a particular button.

Publish-subscribe is similar to message passing in that the publish-subscribe message recipient is not expected to reply to the message. It is different from message passing in an important way: in message passing, the message sender determines who the message recipient is, and the message recipient has no say about what messages it will receive. In publish-subscribe, the message recipient determines what message service it will subscribe to and what messages it is willing to receive. From the point of view of message recipients, the disadvantage of message-passing is that the recipients can receive unwanted messages (such as "spam"); the disadvantage of publish-subscribe is that a recipient may miss messages of interest because it hasn't subscribed to them.

If there is only one publishing agent A_1 in the publish-subscribe scenario, that agent may also serve as the message service, in which case that agent is also the source of all subscribed messages.

1.2.4 Events in Systems

Each of the three types of agent interactions described above involve events.

In request-response, there are potentially four events: (1) the act of sending the request by agent A; (2) the receipt of the request by agent B; (3) the act of sending the reply by agent B; and (4) the receipt of the reply by agent A. For synchronous request-response interactions, especially those that occur over short periods of time, these four events are normally all combined together and considered one event.

In message passing, there are only events (1) and (2) as described above; again, these two events are normally considered one event if they occur over a short period of time.

In publish-subscribe, we will consider the posting of a message by one of the publishers as an event, and the receipt of a posted message by one of the subscribers as an event. Again, if posting and receiving of the message occur over a short period of time, we may consider them as one event.

1.2.5 System state

A system's *state* is a complete description of the system at some point in time. As the agents in the system carry out their defined activities and interact with other agents through events, the state of the system changes.

For example, consider our solar system – our sun, planets, moons, etc. The state of this system at any point in time is the exact position of each of these entities and their velocities with respect to each other. As time passes, these entities will change position in a mostly predictable way. (We say "mostly" because external events such as the nearby passing of an extrasolar mass could affect planetary motion.) The solar system has been studied intensely for hundreds of years, and its predictable behavior has allowed astronomers to identify when events such as eclipses will occur.

As another example, consider a running Java program, which we will call a *process*. The state of the process is the collection of values of all the memory cells that the process depends on or has access to, including program variables, processor registers, RAM, and disk. We assume that all such values can be represented as a finite sequence of binary bits (zeros and ones). As time passes, these values will change in a mostly predictable way. (We say "mostly" because external events such as user mouse clicks can affect program execution.) Millions of computer programs have been written to carry out useful activities, and their predictable behavior has allowed us to trust these programs to work as they have been designed.

1.2.6 Event based systems defined

An *event based system* is a system in which interactions among the agents in the system are governed by events, principally those interactions that are request-response, message-passing, or publish-subscribe.

1.2.7 Discrete systems and events

A system is said to be *discrete* if every possible state of the system can be described using a finite amount of memory (a finite number of bits) and if, over any finite time interval, the state of the system changes a finite number of times. This means that for every state of a discrete system, there is always a *next state*.

- A traffic light is a simple example of a discrete system. There are a limited number of states: green, yellow and red. Over a finite time interval, the light changes a finite number of times.

- By contrast, the solar system is regarded as a continuous system. Each planet moves around the sun in a continuous motion, not moving from state to state.

Computer programs can model either discrete or continuous systems, e.g. we can write a program to model a traffic light or the solar system. Computers operate discretely, however. A computer has a finite amount of memory and executes programs one instruction at a time, moving from state to state.

Since firing an event in a discrete system can result in only a finite number of states, we normally consider events in such a system as discrete as well. A *discrete event system*, then, is a discrete system that is event based.

If a discrete system is *closed* – that is, if there are no possible interactions with anything outside of the system – the state of the system at any one point in time is sufficient to determine all future states. Such a system is also called *deterministic*. However, virtually all interesting discrete event systems are not closed. These include any systems that involve human interaction or information from real-time data acquisition sources.

Our examples in most of the remainder of this book will be discrete event systems, although some theoretical discussions will apply to arbitrary event based systems.

1.2.8 Examples of events

We give several examples of events, highlighting the events of interest in **boldface**. (These examples include more events than we choose to focus on.)

Put on the brakes

The driver of an automobile in heavy traffic **puts on the brakes**, and the automobile's **brake lights illuminate**. This event is observed by motorists traveling close behind. Observing motorists may handle the event by putting on their own brakes, depending on how close they are to the braking automobile in front. These events can propagate backwards down the highway, in a series of *cascading events*.

Spring equinox

An **equinox** occurs in the northern and southern latitudes when the number of hours of daylight and darkness are the same. The spring (or vernal) equinox is preceded by shorter days and longer nights. Ancient agricultural societies may have observed this event using solar "calculators" (*e.g.*, Stonehenge) and used it to plan their growing season.

Fire!

A **person yells "fire!"** in a crowded theater. The occupants of the theater react by **rushing to the exits**.

Stock prices

An on-line investment service updates and **displays the prices** for securities traded in the open market. An investor watches these updates and waits for the price of a particular security to exceed a threshold (a specified amount per share). When the investment service lists the security with **price exceeding this threshold**, the investor **trades shares in the security**.

Alarm clock

A person going to sleep in the evening **sets an alarm clock for some time the next morning. The alarm goes off, and the person awakes.**

Election

The United States holds an election every four years for President. **Voters go to the polls** on election day in November to elect a President. The elected **President takes office** in January.

Digital watches

Many modern electronic devices are event based, including digital watches. Such a watch may have several separate buttons on it. **Pressing a button** sends an event to the watch, for example, to start or stop the watch's timer.

Traffic lights

Traffic lights change from red to green based on timers and sensors. When the **light changes to green**, the **vehicles start moving**.

GUIs

GUIs are almost always event based. They are controlled by **moving and clicking the mouse**, and **by pressing keys**. The program responds by executing the appropriate method for each event.

Interrupts

Computer peripherals interface with the operating system via interrupts. A peripheral device **raises an interrupt** that sends message to the processor telling it that the device needs attention. The operating system responds by executing the interrupt handler.

DB triggers

Database management systems (DBMSs) implement triggers to help maintain the integrity of databases. A trigger is procedural code that is run automatically in response to an event, which may be as simple as inserting or modifying data in a table. For example, **inserting a new row into a table** may cause the table to grow past a predefined threshold, which then requires the DBMS to rearrange the data to a more efficient form. The database management system responds to the event by taking action to correct the problem.

Middleware services

Middleware is computer software that facilitates the development and deployment of distributed programs across heterogeneous computing systems. Middleware provides services that allow messages to be easily passed between the systems regardless of the hardware or operating system of each. The systems are unaware of each other except for the **messages that they send and receive**.

Discrete event simulation

Discrete event simulation is the probably the oldest field of computing that explicitly recognized the central role of events. In discrete event simulations a model of a real world system is simulated over time using a computer. For example, the model might be of a grid of streets with traffic lights and stop signs at various intersections. The simulation allows city planners to test various timings for the traffic lights, or simulate the effect of adding a new light at a particular intersection. The goal is to maximize traffic flow and safety within the system. In discrete event simulation, an **event takes place at simulated time** t, causing other events to be scheduled at a future time $t + \delta t$.

1.3 Attributes of Event Based Systems

Most of this text concentrates on developing event based systems. However, these systems differ from other computer systems in more ways than just their programming. Understanding these differences gives us a better appreciation of what event based systems are about.

1.3.1 State Based

Event based systems are *state based*. The system stays in a stable state until an event occurs. Processing the event changes the state, then the system quiets down and waits for more events. For example, a text editor sits there, not changing the document, until the user presses a key. Then it records the key pressed into the document and waits for the next event. Similarly, an engaged cruise control system in your car keeps the car running at a near constant speed until the driver steps on the brake, or dis-engages the system via controls on the dashboard or steering column.

Conceptually, we divide state into two types, *data state* and *control state*. The *data state* of a system is the collection of variables that the system maintains. For example, in our text editor, the data state contains the document being worked on and possibly some ancillary variables like the document's file name. The document is updated as the user types. The file name is updated only when the user chooses `Save As`.

The *control state* is the collection of variables that determine how the system responds to incoming events. For example, `vi` (one of our all time favorite text editors) has **insert** mode and **command** mode. In **insert** mode, if the user types <shift>ZZ, `vi` inserts two capital Zs at the cursor's current location. In **command** mode, <shift>ZZ directs `vi` to save the document and exit. The events, the <shift>ZZ, are the same in both cases, but the system responds to them very differently.

Embedded systems use computers to control another device. For example, a cruise control system on a car is an embedded system. The computer receives input signals from sensors and outputs control signals to the device. The control state of the embedded system determines the output values. For example, when the cruise control system is engaged the car's electronics send signals to the throttle maintaining the desired speed.

Ideally, control state and data state are disjoint. That is, the editor's mode and the document are not related. The mode has to do with how the text editor behaves. The document is whatever the user is typing. In practice there may be some murky overlap. For example, some text editors will not save an empty document. They treat an empty document differently than one that contains even one character. That first character is part of both the data state and the control state.

1.3.2 Nondeterminism

Nondeterminism means that it is impossible to determine exactly how a computation will proceed. Even given the same inputs the path the computation follows may vary from run to run. All event based systems contain varying degrees of nondeterminism.

Consider any modern desktop operating system. Hundreds or thousands of events occur every second. The mouse moves. Keys are pressed. Network packets arrive. A disk drive signals that data is ready. The user plugs in a USB device. All are events. The entire operating system needs to continue to work solidly, responding to each event, regardless of the order they occur.

The distributed nature of many modern event based applications complicates the matter further. These systems contain a high degree of nondeterminism. Consider a web based flight reservation system. Users from around the world make travel reservations. If multiple users compete simultaneously for the same two seats on a flight, one of them should get both seats. Which user gets the seats is determined by the order in which their events are handled, but this order depends on many unknowns, such as the network load between each user and the system. The system mustn't hang because of the competition. Nor should it give one seat to one user and the other seat to another user.

1.3.3 Loose Coupling

One of the strengths of event based programming is that it allows complex systems to be built from diverse, *loosely coupled* components. The components communicate with each other via events.

Again, consider a desktop operating system. New peripheral devices (printers, network cards, etc.) require device drivers be installed to interface the device with the operating system. Many device drivers are loosely coupled to the kernel of the operating system, as they are developed separately and registered with the operating system while it is running.

Similarly, consider GUIs. The components that go into a GUI are supplied by the language or library being used. When a GUI program starts, there is typically a "building" phase, where the components are placed in the window and the code behind each is registered with the component. This is also a loosely coupled model, as the components were developed independently, code was registered at runtime, and the components execute the code by firing events.

As a final real life example, consider a jet fighter aircraft. The plane is almost certainly event based. The cockpit is full of all sorts of buttons and switches, each of which generates events. The plane's electronics is made up of multiple subsystems. There is the system controlling the engines, the navigation system, the communications system, the radar system, and the weapons system, to name just a few. These subsystems work together and communicate with each other to keep the aircraft functioning correctly, but each exists independently of the others and can be repaired or upgraded, as needed.

1.3.4 Decentralized Control

In days of yore, when the authors were just wee lads, there was a main program. The main program called a subroutine that called another subroutine that called still another subroutine. Each subroutine returned when finished, and the main program proceeded to call its next subroutine. The world had order. The main program was in charge and everybody lived happily ever after, until now $\cdots$

Event based systems use *decentralized control*. The system starts up and waits for events to occur. Each event causes changes to the system. Even a single event can have cascading effects that propagate throughout the system. Nobody is in charge. In many languages the main program is still responsible for starting the system running, but after that, the flow of control depends on events.

Object-oriented programming (O-OP) provides a good fit for designing control of event based systems. O–OP places emphasis on developing highly cohesive objects and methods. Each object represents one thing. Each method does one task and does it well. Event handling code is also very cohesive. The event handling code associated with a GUI component is seldom very long, as each component plays a small role in the overall processing.

1.4 The Event Based Programming Paradigm

What makes event based programming different from other types of programming? Event based programming is paradigm. It is a way of thinking about problems and their solutions. It provides *abstractions* . Languages are known as procedural or object–oriented because they map the abstractions onto language structures. Event based programming is no exception. The event model is its primary abstraction.

1.4.1 The Event Model

At the heart of the event based paradigm is the concept of an *event*. Three types of computational objects are associated with each event, the *event source*, the *event object*, and one or more *event handlers* sometimes also referred to as *event listeners*.

- *Event Sources*
  The event source is the originator of the event. We say that the event source **fires** an event, when it creates an event object and prepares it for the handlers. In our earlier computing examples we tended to emphasize event sources that were hardware: the mouse, the keyboard, or a disk drive. Event sources can be any object, hardware, software or firmware, however. For example, it is not unusual to have a data object in a system, say a list of names, that fires an event when the list changes. There may be multiple other objects that update their state when the event is detected.

- *Event Objects*
  An event object encapsulates the critical data associated with the event. For example, there are few events as ubiquitous as mouse clicks. The data associated with each click includes: the screen or window coordinates of the click, and the button that was pressed. Any handler listening for the click may need some or all of this information, so any reasonable implementation of the mouse clicked event object, regardless of the language or system, will include it.

- *Event Handlers*
  Event handlers respond to events by carrying out the actions specified by the programmer. The handlers are the glue that binds together the other objects to form a working program. Handlers are registered with their event sources. After that, they wait passively for the source to fire an event to them, at which time they run their handling code. Event handlers can contain arbitrary code, but a typical event handler updates the data state and the control state of the system, and possibly notifies other objects of the changes.

1.4.2 Events versus Method Invocations

When we discussed loose coupling in the previous section, we were primarily referring to the relationship between the event sources and their event handlers. The relationship is based on events, not method invocations. The ideas behind the two are similar, but there are several important differences. The event source and event handler are much more loosely coupled than objects in a standard caller/callee relationship.

- Event handlers are registered (and possibly de-registered) with event sources at runtime. This *late binding* means that it is possible to plug–in different handlers with the same event source at different times during execution. This is a type of runtime polymorphism found in many modern languages, but is quite different from the compile and link time semantics of traditional method calls.

- There may be zero, one or multiple handlers registered for an event. Sending an event to multiple handlers, also known as *multicasting*, proves useful when there are multiple views that rely on the same data. For example, consider the case of a hospital information system. A doctor, a nurse, and a pharmacist each have views of a patient's records. If the doctor updates the patient's chart, it is important that the nurse's view and the pharmacist's view both be updated appropriately. Having the patient (or chart) object multi cast the event is an appropriate solution to this problem.

- Event handlers do not return any information to the event source. That is, by the nature of the paradigm, event handlers have a `void` return type.

- Multiple event sources may fire events to the same handler. This type of *multiplexing* is frequently seen in GUIs when there are multiple ways to accomplish the same task. For

example, a program might have a menu item `File -- Save` and might also have a `Save` icon. The same handler is registered with both. This simplifies maintenance and debugging, as there is only one place the code needs to be updated.

- In most event based languages, the event source does not block, waiting for the handlers to complete. It fires the event, then continues to run. In the terminology introduced earlier in the chapter, we say the event source and event handler execute *asynchronously*.

- There may be a delay between when the event fires and when each handler processes it. This delay may be caused by a backlog of other events that are awaiting processing, other handlers executing for the current event, a network delay if the handler is on a remote system, or numerous other reasons. The key point is that the delay can occur.

As with any paradigm, languages and libraries that implement support for event based systems vary. Each of the properties listed above may or may not be true. For example, in Java, some events are handled asynchronously, while others are handled synchronously. It is important that you understand the tools you are using to develop event based systems, as even a small change in the semantics can have major implications for the program's correctness.

1.4.3 Writing Event Based Programs

Event based programs are generally a blend of more traditional code, either procedural or object–oriented, and event code. The event based portion of the code is typically for a subsystem, like GUI I/O, or database access. Methods and data structures in the traditional code will be accessed from the handlers, so well designed, cohesive code throughout is critical.

Event based programming languages have hundreds of supporting classes and interfaces in their libraries. Each has a particular purpose, but gaining a working knowledge of the libraries can be a daunting task. Becoming a good Java or C# event based programmer takes time and patience.

Modern IDEs, like MSDev .Net, NetBeans, Eclipse and JBuilder, come with visual programming tools. These IDEs make naive programming of GUIs simple. The user drags and drops components into a window. Event handler stubs are automatically generated by the system. All the programmer has to do is fill their method bodies. On the other hand, as we shall soon see, there is much more to event based programming than putting a bit of code behind a button.

1.5 Goals of the Text

The remainder of the text has three goals:

- We will begin our study by taking a look at event based programming in Java. While there are many languages that support event based programming, we couldn't hope to present them all, and Java's event model is fairly clean and easy to work with. We will study the event related classes and learn how to write event handlers that work with them. We will also look at how to specialize Java library classes to meet our own needs.

- Java is certainly not the only system that supports event based programming and we explore a variety of other languages and application areas that embrace events. These include: hardware and operating systems, web services, and distributed systems. You will not become an expert in any of these fields by reading this text, but you will gain an understanding of the unifying concepts that arise from the event based paradigm.

- Throughout our discussions we will occasionally step back and ask what tools are available to help us design our systems. We will look at UML models and other formalisms that help us document our design. We will also look at the design patterns that apply to event based programming. Finally, we will look at distributed systems modeling tools that help us address the synchronization problems that arise.

Chapter 2

Event Based Programming Basics

2.1 Introduction

This chapter introduces you to the fundamentals of event based programming. Obviously, if we are going to discuss programming, we will have examples, and those examples will be implemented in some language. We chose Java. Java has a clean event model where each class or interface plays a particular role and where the various components work together nicely to form an entire application. Since, Java is (mostly) platform independent, you may work through all our examples under Windows, or Linux, or on a Mac – on almost any desktop computer.

Graphical User Interfaces (GUIs) are also a good way to introduce the nuts and bolts of event based programming. GUI components such as menus, check boxes, radio buttons, and text boxes communicate with an application via events. The purpose of this chapter is not to study GUI programming in depth. GUI programming deserves a complete textbook, and there are already many good texts on the topic [Johnson, 2000, Walrath et al., 2004]. Our purpose is to illustrate various aspects of event based programming. We use GUIs as the domain.

Java includes two packages for GUI development, the *Abstract Window Toolkit (AWT)* and *Swing*. Our examples use Swing. Swing is the newer package and the more popular of the two. By the time you finish this chapter, you will be familiar with some of the basic Swing classes, including: `JFrame`, `JButton`, `JLabel`, `JTextfield`, `JScrollPane`, `JList`, and `JPanel`, as well as the Swing menu classes. You will also understand the events that Swing components use to interact with the application. You will not have mastered Swing, however. If you are interested in pursuing Swing programming more thoroughly, you are encouraged to pick up a good Swing programming text [Elliott et al., 2002, Deitel and Deitel, 2007, Deitel et al., 2002].

All our examples are relatively short and may be coded using any text editor. However, all source code examples may also be downloaded from the text's web site.

2.1.1 Integrated Development Environments

If you intention is to build professional quality GUIs quickly, the authors recommend using an Integrated Development Environment (IDE). There are several good Java IDEs, *NetBeans* and *Eclipse*, for example, and a variety of plugins that allow programmers to develop Swing programs using drag and drop techniques. These are not discussed here, as they hide many of the details of event based programming in which we are interested.

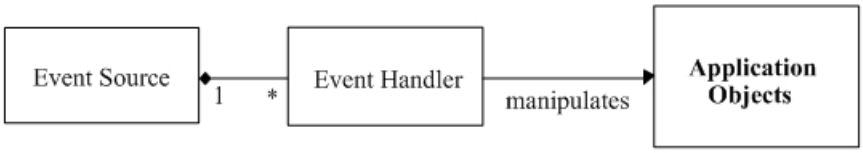

Figure 2.1: The object–oriented event processing model

2.2 The Object–Oriented Event Model

The fundamental building blocks of object–oriented programs are classes and objects. Object–oriented event based programs are no exception. The event source (an object) *fires* an event which is processed by the event handlers (other objects). Each handler processes the event by manipulating application objects. Figure 2.1 shows the relationship among the various pieces.

As we will see in later chapters, designing event sources poses some complex issues. We will avoid these issues in this chapter by letting the Swing classes serve as the sources. Swing components are the programmatic representations of the sources for *input events* such as mouse clicks and keyboard presses. Of course, a particular physical device (mouse or keyboard) is really the "source" of the event, and that this low level event propagates upward to the Swing application. We pick up the event processing when the Swing component realizes that the event has occurred. For more on the upward propagation of events, see Chapter 5.

The event handlers, which execute when the event fires, are application specific. That is, each button in each application has handlers designed explicitly for it. Assuming that you have already implemented the data structures and other application classes with which your GUI will interact, coding a Swing GUI becomes a three step process:

1. Identify the GUI components to be used and lay out the GUI interface.
2. Code the event handlers that go behind the GUI components[1].
3. Register the handlers with the GUI components.

2.2.1 A Simple Example

A simple example can serve to clarify these ideas by putting them into a concrete context.

In this example there is a single button. When the button is clicked, a message box is opened. The main GUI interface is shown in Figure 2.2.

Figure 2.2: The user interface for `ClickMe.java`

The resulting message box is shown in Figure 2.3.

[1]A GUI component such as a button has a visible attribute that appears to a user on the screen, and clicking on the button activates program code (the handler) that is invisible to the user. We think of the button as hiding the handler, or that the handler is "behind" the button, away from the view.

```
1 /** ClickMe.java
2     This is a very simple event driven Java program.
3     It contains a single button, that when clicked pops open a message dialog.
4
5     Written by: Stuart Hansen
6     Date: September 2008
7 **/
8
9 import javax.swing.*;
10 import java.awt.event.*;
11
12 public class ClickMe extends JFrame {
13    JButton button;  // Our one and only button
14
15    public ClickMe () {
16       // The button is our event source
17       button = new JButton("Click me");
18
19       // We register a handler with the source
20       ActionHandler handler = new ActionHandler();
21       button.addActionListener(handler);
22
23       // We add the button to the viewable area of the window
24       getContentPane().add(button);
25
26       // We set a couple of window properties and then open the main window
27       setSize(100, 100);
28       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29       setVisible(true);
30    }
31    // A very simple main method
32    public static void main (String args[]) {
33       new ClickMe();
34    }
35 }
36
37 // The handler class.
38 class ActionHandler implements ActionListener {
39    public void actionPerformed (ActionEvent e) {
40       JOptionPane.showMessageDialog(null, "Ouch! That hurt.");
41    }
42 }
```

Lines	Commentary
9, 10	The `javax.swing` package includes the Swing classes for windows, buttons, textboxes, etc. The `java.awt.event` package is also needed, as it includes many of the event classes.
12	The class `ClickMe` inherits from `JFrame`. `JFrame` is Swing's window class. By inheriting from it, our program gets a main window in which we develop our GUI.
13	The `JButton` is the only GUI element in our window.
15–30	The `ClickMe` constructor in which we instantiate our objects, register the handler with source and set a few properties.
17	This instantiates the button.
20	`handler` is an `ActionHandler` object. The handler is responsible to responding to button clicks. `ActionHandler` is a separate class, whose source code is at the end of the file.
21	`handler` is registered with the button at runtime. Note that this is an example of *polymorphism* or *dynamic binding*, a central theme in event based programming.
24	The button is added to the `JFrame`. Swing requires that we add the button to the `contentpane` of the application rather than directly to the `JFrame`. This separates the handling of the GUI components from the other windowing responsibilities, e.g. closing and resizing.
27	The `JFrame` is sized to 100x100 pixels.
28	The application should exit when the `JFrame` closes. If this line is omitted, you may close the GUI window, but the application will continue to run in the background.
29	The `JFrame` is set to be visible.
32–34	The main method. As with many GUI and other event driven programs, the main method's body shrinks to a single line of code that instantiates the application object. In our application line 33 constructs a new `ClickMe` object.
38–42	The `ActionHandler` class. It contains one method, `actionPerformed()` which is invoked when the button is clicked. The method opens a message window.

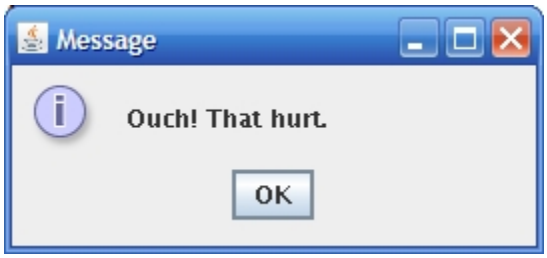

Figure 2.3: The message box opened by clicking the button.

Many beginning Java programmers do not realize that it is legal, and many times desirable, to place multiple Java classes within one file, as we done here. The only restriction Java places on us is

that at most one class may be `public`: this must be the class containing the `main()` method. In our program we place the `ActionHandler` class within the `ClickMe.java` file, because it is only used by this application.

The `ActionHandler` class implements the `ActionListener` interface. Objects implementing this interface must have a method with the particular signature:

```
public void actionPerformed (ActionEvent e)
```

Since our handler is registered with the button (line 21), this `actionPerformed` method is called whenever the button is clicked. Our particular `actionPerformed` method displays a message dialog box saying "'Ouch! That hurt."'

While this example is very simple, it illustrates many of the basic features of Java event based systems. It has an event source (the `JButton`) and an event handler (the `ActionHandler`). It *registers* the handler with the source at runtime. The handler contains a method with a specific signature (`actionPerformed(ActionEvent e)`) that is called when the button is clicked. We should also note that the event object, `ActionEvent e`, is always passed to the handler, but in our example `e` is never used.

2.3 Java Language Features to Support Event Based Programming

There are several features of Java that support event based programming. Among them are anonymous classes and inner classes. The next several examples show how these features can be used to make our programs more elegant.

2.3.1 Anonymous Classes

Most Java programming students are familiar with anonymous objects. An anonymous object is one that is instantiated, but never assigned to a variable. For example, in the ClickMe program above, the `main()` method instantiates a `ClickMe` object, which starts the entire program running, but that object is never assigned. It remains anonymous.

Anonymous classes are similar. A class is created, but never given a name. Two conditions should be true before using an anonymous class:

1. There must be only one place in the code where an object of this type is instantiated. We will define the anonymous class and instantiate objects at this location.

2. The anonymous class should only contain one or at most two short methods that are being defined or overridden. If the class is longer, the code will be much more readable if defined as a named class.

Anonymous classes are useful to define event handlers, because handlers generally meet these conditions. Often an event handler is only referenced when it is instantiated and registered with the source (condition 1). Similarly, a handler only needs to define the methods expected by the listener interface. In our above example, this was `actionPerformed()`. Thus, event handlers make ideal candidates for anonymous classes.

ClickMe using Anonymous Classes

Here is the ClickMe program again, using an anonymous handler.

```
 1 /** ClickMeAgain.java
 2     This program illustrates the use of anonymous classes for handlers
 3     Written by: Stuart Hansen
 4     Date: September 2008
 5 **/
 6
 7 import javax.swing.*;
 8 import java.awt.event.*;
 9
10 public class ClickMeAgain extends JFrame {
11    JButton button;  // Our one and only button
12
13    public ClickMeAgain () {
14       // The button is our event source
15       button = new JButton("Click me");
16
17       // Register the handler
18       button.addActionListener(new ActionListener() {
19          public void actionPerformed (ActionEvent e) {
20             JOptionPane.showMessageDialog(null, "I said,  \"Don't do that.\"");
21          } }
22       );
23
24       // We add the button to the viewable area of the window
25       getContentPane().add(button);
26
27       // We set a couple of window properties and then open the main window
28       setSize(100, 100);
29       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30       setVisible(true);
31    }
32
33    // Event driven program regularly have simple main programs
34    public static void main (String args[]) {
35       new ClickMeAgain();
36    }
37 }
```

Lines	Commentary
18–22	An anonymous instance of an anonymous handler. We construct an `ActionListener` followed by a pair of braces, { and }, and we define the handler methods within them. In this program, the only method defined this way is `actionPerformed()`. Note that `ActionListener` is an interface, but because we define `actionPerformed()`, we can instantiate it. Anonymous classes can also be derived from other classes. This is useful if we want to specialize a previously defined handler by overriding one or two methods.

The remainder of the code is virtually unchanged from the previous example. The only difference is that the named class, `ActionHandler`, is gone – replaced by the anonymous class shown above.

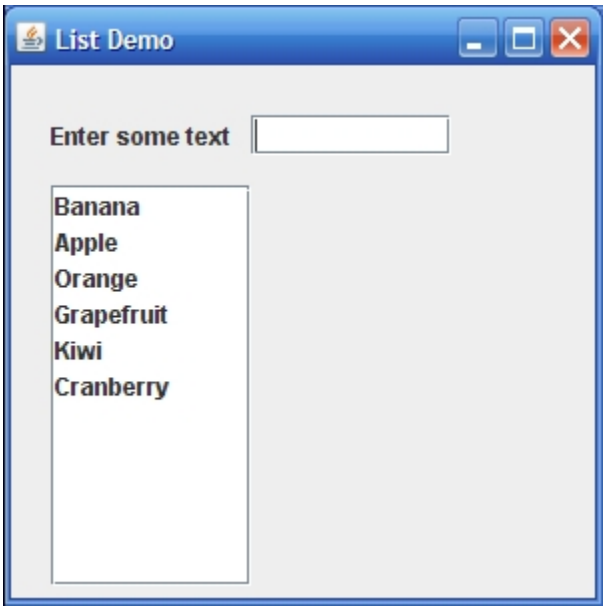

Figure 2.4: The GUI for the List of Strings program.

2.3.2 Inner Classes

Many modern object–oriented languages, including C++, C# and Java allow a programmer to define *inner classes*. These are classes defined inside of another class. Inner classes are useful in several different situations. If class A takes complete responsibility for all instances of class B, it is appropriate to define class B as a private inner class, inside of A. This prevents other classes from attempting to access B directly. For example, linked lists contain nodes for which the list is completely responsible. Nodes should not be seen outside of the list. They serve no purpose outside of the list. We can define the `Node` class a private inner class to the `LinkedList` class. The linked list will have complete access to the nodes, but nodes will be completely invisible outside the list.

Inner classes also provide improved scoping. The *scope* of a variable or method is where it is visible. In object–oriented languages, instance variables and methods are declared `private`, `public`, or `protected`. This determines the variable's visibility. Good software engineering practice tells us to declare our variables as `private` so that they are only visible within the class where they are declared. We then grant access to them via `public` methods that manipulate them only in controlled ways.

In modern object–oriented languages, instances of an inner classes have access to all members of the encapsulating class, including private members. This means that an event handler defined as an inner class can call private methods and access private data in the encapsulating/outer class, as needed.

2.3.3 List Example using Inner Classes

This example illustrates inner classes, as well as several additional Swing classes. The objective of the program is to display a list of strings that the user enters. The GUI is shown in Figure 2.4. The user types text in the input field following the prompt. When she presses enter, the text is added to the list.

```
 1 import javax.swing.*;
 2 import java.awt.*;
 3 import java.awt.event.*;
 4
 5 /* ListDemo illustrates the use of JLabels, JTextFields,
 6  * JScrollPanes and JLists.
 7  *
 8  * Written by: Stuart Hansen
 9  * September 2008
10  */
11 public class ListDemo extends JFrame {
12    // These elements form the GUI of the application
13    private JLabel enterLabel;
14    private JTextField enterField;
15    private JScrollPane listPane;
16    private JList list;
17
18    // The constructor
19    public ListDemo () {
20       super ("List Demo");
21       enterLabel = new JLabel();
22       enterField = new JTextField();
23
24       // The DefaultListModel on the next line gives us the
25       // ability to add to the list
26       list = new JList(new DefaultListModel());
27
28       // The list is added to a JScrollPane, so that we may
29       // view lists larger than the display
30       listPane = new JScrollPane(list);
31
32       // setup the contentPane to use absolute coordinates
33       Container contentPane = getContentPane();
34       contentPane.setLayout(null);
35
36       // initialize and add the components to the GUI
37       enterLabel.setText("Enter some text");
38       enterLabel.setSize(100, 30);
39       enterLabel.setLocation(20, 20);
40       contentPane.add(enterLabel);
41
42       enterField.setSize(100, 20);
43       enterField.setLocation(120 ,25);
44       contentPane.add(enterField);
45       enterField.addActionListener(new InputHandler());
46
47       listPane.setSize(100, 200);
48       listPane.setLocation(20, 60);
49       contentPane.add(listPane);
```

```
50
51        // Set the windows size and close operation
52        setSize(300, 300);
53        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
54
55        // display the application window
56        setVisible(true);
57     }
58
59     // The ActionListener for the JTextField
60     // The event if fired when enter is pressed
61     // The text is added to the list
62     private class InputHandler implements ActionListener  {
63        public void actionPerformed(ActionEvent e) {
64           String text = enterField.getText();
65           DefaultListModel model = ((DefaultListModel)(list.getModel()));
66           if (!text.equals("")) {
67              model.addElement(text) ;
68              enterField.setText("");
69           }
70        }
71     }
72
73     // a very simple main program
74     public static void main (String args[]) {
75        new ListDemo();
76     }
77 }
```

Lines	Commentary
13–16	Declare the various Swing components for this example.
13	Declares the `JLabel` that prompts the user.
14	Declares the `JTextField` where the user enters the strings.
15–16	The components on lines 15 and 16 work together. `JList`s are not scrollable, so when using a `JList` it is almost always best to wrap it in a `JScrollPane` before adding it to the application. Scroll bars will then appear automatically if the list grows larger than its viewable area.
19–57	Define the application's constructor.
21–30	Instantiate the various GUI components.
26	Pass an instance of `DefaultListModel` to the `JList`'s constructor. It sounds silly, but `DefaultListModel` is not the default list model for `JList`s, so we need to pass one to the constructor, if we want to use it in place of the real default. In our application, the main advantage of using the `DefaultListModel` is that it makes it easier to add objects to the list.
30	Wrap the `JList` inside of the `JScrollPane` as discussed above.
33–34	Set the `JFrame`'s contentpane and set its *layout manager* to `null`. Layout managers control the location and size of components in a GUI. There are a number of good layout managers, but any discussion of them is beyond the scope of this chapter. Instead, by setting the layout manager to null, we tell the application to use the sizes and locations that we set explicitly in the code.
36–49	Set the sizes, locations and a few other properties of the various GUI components and add each to the `JFrame`.
52–56	Set a few `JFrame` properties and show the application window.
62–71	Define the handler for the `JTextField`. An `ActionEvent` is generated when the user presses enter while the focus is on the field. As in the previous example, this handler implements the `ActionListener` interface. Conceptually, the `actionPerformed()` method is straightforward. We get the text from `enterField`, and if it isn't `null` add it to the list. There are a number of details we should note: • The handler is defined as a private inner class. This means that the class and instance of the class are not visible beyond the `ListDemo` class. • `actionPerformed()` accesses data and methods in the `ListDemo` class. It calls two methods from `enterField` and one method from `list`. These are *private* data members of `ListDemo`, but are visible to the handler because it is an inner class. • All Swing components, including `JList` use a model–view–controller (MVC) architecture. In the next section we discuss MVC in more detail. For now, note that the data is stored in the model portion of the component. Therefore the handler must call `getModel()` before it can add to (or delete from) the list.

2.4 Inheritance

Inheritance is the primary way of reusing existing code while specializing it to the needs of a specific application. A derived class inherits from a base class. It automatically gets all the data and methods already defined in the base class. The programmer can then add more data and methods, and *override* methods that already exist.

Inheritance can play a major role in developing GUIs and other event base programs. Languages come with large GUI APIs that contain buttons and sliders and all the types of components that go into a typical GUI. Each has a standard appearance and user interactions, e.g. a GUI button looks like a button, and computer users all know that you click it to make things happen. By contrast, a label displays information to the user and the user doesn't expect anything to happen if they click on it.

An easy way to build complex GUIs is to extend existing GUI classes so they appear and behave the way you need. Let's consider two examples:

- In our previous Java example, we displayed a list of strings. The list was displayed in the order the strings were entered. A simple specialization would be to maintain the list in alphabetical order.
- As another example, consider developing a drawing program. The user will create a picture by dragging the mouse. The program displays the figure in real time, as it is created.

In both cases, we start with an existing class and extend it for some particular purpose. We will develop Java implementations of each of these after a bit more discussion.

2.4.1 Model–View–Controller

GUI components are developed using a *Model–View–Controller (MVC)* approach. MVC is way of thinking about GUI components and programs that divides their implementation into three parts:

- a model – containing the data,
- a view – visually presenting the data, and
- control – through which other objects and the user interact with the data.

The component wraps the model, view and controller into a single object.

The model, view and controller are tightly coupled, because the component can't exist without all three, but the coupling occurs in very specific ways.

- Control updates the model and on occasion may directly tweak the view.
- The view depends on the model for the data to display.
- Control code, at least in Java, frequently does not exist as an independent object, but consists of the methods and handlers within the component that change the component's state.

The model, view and controller remain loosely coupled in the sense that each one has distinct responsibilities and can be modified or replaced without requiring changes to the other two. Model, view and control each have specific responsibilities, and as long as they live up to their responsibilities their internal functioning is independent of the other two. For example, in the `ListDemo` code above, the `JList` was assigned a new model, the `DefaultListModel`, without requiring any changes to the list's view or control.

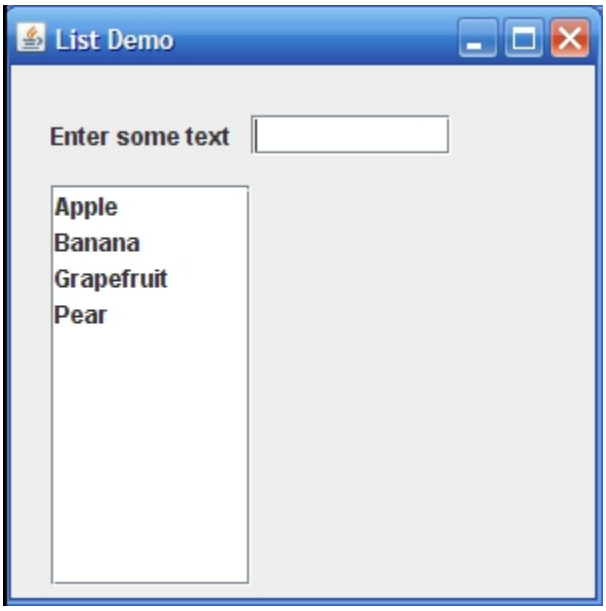

Figure 2.5: The sorted list program during a run.

The Sorted List Example

This example is identical to the previous one, except that the elements in the list are maintained in lexicographic order. In the earlier example we showed how to replace a `JList`'s model with an instance of `DefaultListModel`. In this example we will use a specialized list model of our own design.

`ListModel` is a Java interface. Any object that implements the interface could be used as the model for a `JList`. Our `SortedListModel` extends `DefaultListModel` which, in turn, implements `ListModel`. This way we will only need to override methods of interest. The rest we inherit from `DefaultListModel`. We choose to only override the `addElement()` method, modifying the code so that the list is maintained in sorted order. As noted in the code's comments, better code would also override all other methods that can add data to the list, or modify elements already in the list, but for the sake of brevity this is not done.

```
 1 import javax.swing.*;
 2
 3 /* SortedListModel.java replaces the default list model with one
 4  * that keeps the list is sorted order.
 5  *
 6  * Written by: Stuart Hansen
 7  * September 2008
 8  */
 9 public class SortedListModel extends DefaultListModel {
10    // We override addElement in the DefaultListModel class
11    // so that the JList remains sorted.
12    // Note that in a more complete application, add()
13    // set() and setElementAt() should also be overridden.
14    public void addElement(Object obj) {
15       String str = obj.toString();
16       int index = getSize();
17       while (index > 0 && ((String)elementAt(index-1)).compareTo(str) >=0) {
18          index--;
19       }
20       super.add(index, obj);
21    }
22 }
```

Lines	Commentary
14–21	The new `addElement()` method uses a linear search to find the correct lexicographic location for the element. The while loop starts at the end of the list and moves backwards toward the first element, until it finds the correct location or reaches the list's beginning. The call to `super.add()` is to the `add` method in the `DefaultListModel` which inserts the element at that location.

We do not show the remainder of the code for this example, because only one other line is changed, the line that instantiates the `JList`. A `new SortedListModel()` is passed to the `JList` constructor instead of the `DefaultListModel`. Figure 2.5 shows the modified program during a run.

ListModelEvents

The astute reader will notice that our revised code did not change the events that fired or the control code. Only the model changed. The *external events* of GUI components control the interaction between the user and application. Our modified program did not touch this code. We only changed the model, which modified how the data was stored and indirectly how it was displayed.

How then did the `JList` know that it needed to update its view when an element was added? The answer is that it uses *internal events*. An internal event is one whose source and handler both reside in the same component. Only the model, view and control of the component need be aware of these.

In our example, the list model fired a `ListDataEvent` to the `ListModelListener`s whenever the model changed.

Because our new model inherited all the infrastructure to register and fire `ListDataEvent`s from `DefaultListModel`. Our new `addElement()` method fired the event within its call to `super.add()`. No explicit event firing was needed on our part.

The `JList`'s view listened for the events and updated itself appropriately. This approach also works well if we have multiple views of the same model, or when changes to the model propagate to other non–view objects [2].

A Drawing Application

In this section we present another example of events working with MVC, creating a drawing application. The user draws in a window by pressing the left button and dragging the mouse within the window, see Figure 2.6.

The application is built around a specialized `JPanel`, named `DrawPanel`. Developing the `DrawPanel` provides an excellent example of extending a Swing component with redefined and augmented model, view and control.

`JPanel`s are *containers*. A container is a component whose primary role is to hold other components. That is, programmers add buttons, textboxes and other components to them, and then later add the container to a window/frame. Containers don't have much specialized functionality, making them an ideal base class for extending when creating a new specialized component.

The more general question of which component a programmer should extend to create a new, specialized component depends a lot on the programmer's needs and the language/API being used. The programmer should try to maximize the amount of code that can be reused. If you want to be able to click on the component, a button is an obvious choice. If you want to be able to type text into the component, a textbox is the obvious choice. If you want to define something completely new and different, then choosing a component with minimal pre–existing functionality, e.g. a `JPanel` in Java, makes good sense.

We start by presenting our `DrawPanel`, then show how it can be used in a complete application.

[2] It is possible in Java to register other listeners for `ListDataEvents`. This is illustrated in the List of Doubles example, later in this chapter.

```
 1 import javax.swing.*;
 2 import javax.swing.plaf.*;
 3 import java.awt.*;
 4 import java.awt.event.*;
 5 import java.util.*;
 6
 7 /**
 8  * DrawPanel implements a JPanel that can be drawn on using
 9  * the mouse
10  *
11  * @author Stuart Hansen
12  * @version September 2008
13  */
14
15 // We specialize the JPanel to contain a drawing.
16 public class DrawPanel extends JPanel {
17    // The model used for the JPanel is a list of curves.
18    ArrayList<Curve> listOfCurves;
19
20    // Two additional state variables to aid us in creating the drawing
21    Curve  currentCurve;     // the current curve being drawn
22    Color  currentColor;     // the current drawing color
23
24    public DrawPanel() {
25       super();
26
27       // Initialize the model
28       reset();
29
30       // replace the view
31       setUI (new DrawPanelUI());
32
33       // add specialized control
34       addMouseListener(new MouseHandler());
35       addMouseMotionListener(new MouseMotionHandler());
36    }
37
38    // A setter for the color
39    public void setColor(Color col) {
40       currentColor = col;
41    }
42
43    // a getter for the color
44    public Color getColor() {
45       return currentColor;
46    }
47
48    // reset the drawPanel model
49    public void reset() {
50       listOfCurves = new ArrayList<Curve>();
51       currentColor = Color.BLACK;
52       repaint();
53    }
```

Lines	Commentary
16	Declare `DrawPanel` to extend `JPanel`. By extending `JPanel` we get all the functionality already associated with it. The other advantage of inheriting from `JPanel` is that it also lets us use a `DrawPanel` wherever a `JPanel` would be permitted. This makes it easy to add `DrawPanel`s to `JFrame`s or other containers.
18–22	Declare the model variables for the `DrawPanel`. The drawing is made up of a collection of curved lines (poly–lines), which we store in an `ArrayList`. `Curve` is a private inner class which is discussed below. The other two variables, `currentCurve` and `currentColor`, are userful while creating the drawing. Each time a new curve is started `currentCurve` is updated. `currentColor` is updated when `setColor()` is called.
24–36	The constructor follows a fairly standard pattern for components that extend Swing components. It does base class initialization, then initializes the model, view and control.
25	`super()` initializes the base class, in this case `JPanel`. It is always a good idea to explicitly initialize the base class with a call to `super`. It is required in Java if you want to pass parameters to the base class constructor.
28	`reset()` initializes the data model portion of the component. The model initialization code is placed in a separate method, `reset()`, so that the drawing can also be re-initialized later, not just when the application is started.
31	`setUI()` sets the user interface to our specialized view. `DrawPanelUI` is discussed below.
34–35	Mouse handlers are added specifying how the mouse will be used to make a drawing. `MouseHandler` and `MouseMotionHandler()` are discussed separately below.
39–46	`setColor()` and `getColor()` are self explanatory.
49–53	`reset()` re-initializes the model portion of the `DrawPanel`. The `listOfCurves` is set to an empty list and the `currentColor` is reset to black. The call to `repaint()` directs Java to repaint the `DrawPanel`. Repainting in Swing is asynchronous, much like event handling. That is, `repaint()` does not do the repainting. Instead, it directs Java to repaint the component at its earliest convenience. There are ways to force immediate repainting, but `repaint()` is almost always the more appropriate method to call. Note that this approach is distinct from the event based approach used in the previous example. We do not fire a `JPanelDataEvent` as that event does not exist. The result of calling `repaint()` is similar, however. The `DrawPanel` is redrawn on the screen.

The `DrawPanel` class is short because it delegates the responsibility for doing much of the work to private the model, view and control. For example, a programmer using the `DrawPanel` class does not need to know how a curve is represented, so the `Curve` class is naturally a private inner class. Similarly, the low-level details of displaying the drawing are best kept private. Again, a private inner class is ideal.

Modeling a Curve

`Curve` is a private inner class to `DrawPanel`. It is used to model individual curved lines in the drawing. Each curve contains a color and a list of points on the curve.

```
55    // Each curve has a color and a list of points.
56    // The points form a series of line segments, so it is really a poly-Line,
57    // not a true curve.
58    // We use an inner class to model the curve.
59    private class Curve  {
60       private Color color;                  // the color of the curve
61       private ArrayList<Point> points;      // the points on the curve
62
63       // the constructor initializes the color and the list
64       public Curve (Color c, Point p) {
65          color = c;
66          points = new ArrayList<Point>();
67          points.add(p);
68       }
69
70       // get the Color
71       Color getColor() {
72          return color;
73       }
74
75       // returns an iterator over the points
76       public Iterator<Point> iterator() {
77          return points.iterator();
78       }
79
80       // adds a Point
81       public void add (Point p) {
82          points.add(p);
83       }
84    }
```

60–61 The `Curve` class contains two data elements, the curve's color and a list of points. Joining the points forms a series of very short line segments, a.k.a. a poly–line. The segments are so short, however, that the curve appears smooth to the naked eye when rendered.

64–68 The constructor initializes the color and the `ArrayList`. Because it creates a new curve when the mouse button is pressed, there will be one point in the curve when it is constructed, the location where the initial button press occurred.

70–83 All of the methods in the class get or modify data values. This is very common in model classes, as their primary purpose is to hold the data.

Populating our model with curves is the business of the control code, discussed later.

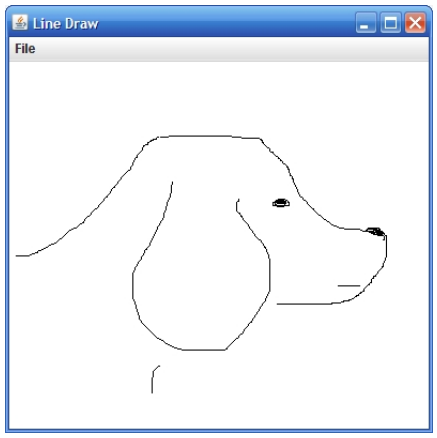

Figure 2.6: An example of a `DrawPanel`. The image was created by holding down the left mouse button and dragging the mouse.

Replacing the View

There are several ways to update the view of a component in Java. An elegant and object–oriented way is to update the view object, a.k.a. the `ComponentUI` for the component. Swing delegates responsibility for rendering a component to its `ComponentUI` object. To create a new view, we specialize `ComponentUI` and override its `paint()` method. As seen in the constructor above, we then assign the component a new `ComponentUI` using the `setUI()` method.

```
86    // The DrawPanelUI class knows how to display the drawing
87    private class DrawPanelUI extends ComponentUI {
88       public void paint (Graphics g, JComponent c) {
89          // We iterate across the list of curves, drawing each
90          Iterator<Curve> curveIterator = listOfCurves.iterator();
91          while (curveIterator.hasNext()) {
92
93             // We iterate across each curve drawing it
94             Curve curve = curveIterator.next();
95             Iterator<Point> pointIterator = curve.iterator();
96
97             // Set the color for this curve
98             g.setColor(curve.getColor());
99
100            // Iterate across the Points rendering the line segments
101            Point p1 = pointIterator.next();
102
103            while (pointIterator.hasNext()) {
104               Point p2 = pointIterator.next();
105               g.drawLine((int)p1.getX(), (int)p1.getY(),
106                          (int)p2.getX(), (int)p2.getY());
107               p1 = p2;
108            }
109         }
110      }
111   }
```

This is the most complex code of the entire application. The drawing is rendered using nested loops. The outer loop iterates across all the curves in the drawing. Each curve is rendered by first setting its color and then iterating across its points drawing the poly–line as we go.

88	Override `ComponentUI`'s `paint()` method. It takes two parameters, a `Graphics` object and a reference to the component we are painting. The details of working with `Graphics` objects are beyond the scope of this text. The class contains over 40 different methods, most of which are related to rendering.
98, 105–106	Our `paint()` method only uses two methods from `Graphics`, `setColor()` on line 98, and `drawLine()` on lines 105–106. The semantics of each of these should be intuitively clear. Further documentation on the `Graphics` class can be found in the Java API documentation.

Note that there are several places in the view code that directly access the `DrawingPanel`'s model, both when getting the iterators and when invoking `setColor()` and `drawLine()`. Again, because `DrawPanelUI` is a private inner class, it has direct access to these data and methods.

Mouse Input Handlers

Our final private inner classes are the event handlers. Java separates mouse events into those associated with moving the mouse and those associated with pressing mouse buttons, so we need two handlers for the mouse input.

An *event adapter* implements all the methods of an event listener with empty method bodies. The notion of an empty method might strike some as strange. The method is called, does nothing and returns. However, adapters are useful because a handler may inherit from the adapter class and override only the subset of the methods needed for the particular application. Figure 2.7 shows how adapters fit into the basic event handling architecture.

Some event sources only fire one type of event, e.g. a `JMenuItem` only fires an `ActionEvent`. In this case, the interface only contains one method declaration and there is no need for an adapter because there are no "extra" methods.

The need for adapters arises because the Java event classes sometimes represent multiple closely related events. For example, the `MouseEvent` class represents a number of different mouse related events, including: mouse entered, mouse exited, mouse pressed, mouse released and mouse clicked. The `MouseListener` interface contains a method declaration for each of these. If a component is only interested in mouse clicked events, its mouse handler inherits from the `MouseAdapter` class and overrides the `mouseClicked()` method. When a mouse clicked event is fired by the source it is handled by the mouse handler. Other mouse events are passed up the inheritance hierarchy and handled by the mouse adapter's empty methods.

Both the `mouseListener` and `mouseMotionListener` interfaces specify several methods, so we use adapter classes for both our handlers.

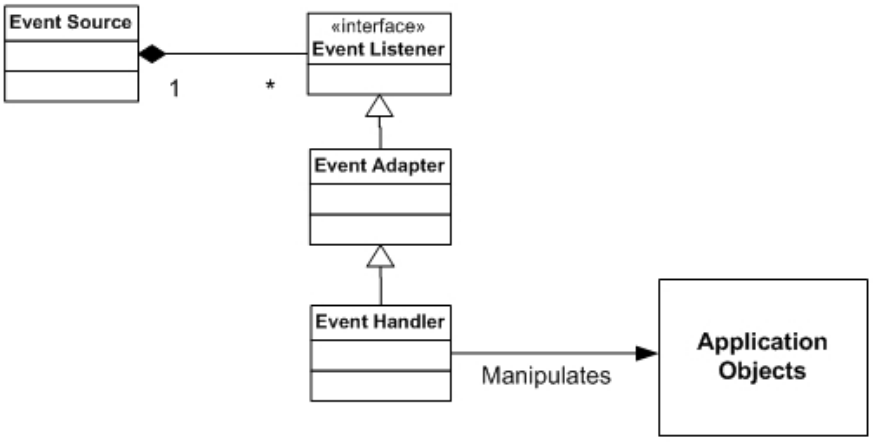

Figure 2.7: Java achieves decoupling between the event source and event handler by placing an interface and an adapter between them. The event listener specifies the methods that the event source expects in all handlers. The event adapter implements all of the methods of the interface with empty method bodies.

```
113    // The following event handlers are part of the JPanel's control
114    // This handler adds points to the current vector
115    private class MouseMotionHandler extends MouseMotionAdapter {
116       public void mouseDragged (MouseEvent e) {
117          if (SwingUtilities.isLeftMouseButton(e)) {
118             currentCurve.add(e.getPoint());
119             repaint();
120          }
121       }
122    }
```

While the mouse is dragged, we add points to the curve.

115 declares our handler to be a subclass of `MouseMotionAdapter`. `MouseMotionAdapter` defines empty methods related to moving the mouse. We just override the one in which we are interested `mouseDragged()`.

116–121 `mouseDragged()` is called when the user holds down any mouse button and moves the mouse. We only want to draw if it is the left mouse button, so the code includes an `if` statement checking this condition. The result is that points are added to the current curve when we drag with the left button pressed.

```
123     // When the mouse is first pressed a new curve is started
124     private class MouseHandler extends MouseAdapter {
125        public void mousePressed (MouseEvent e) {
126           if (SwingUtilities.isLeftMouseButton(e)) {
127              currentCurve = new Curve(currentColor, e.getPoint());
128              currentCurve.add(e.getPoint());
129              listOfCurves.add(currentCurve);
130           }
131        }
132     }
133 }
```

124 `MouseHandler` extends `MouseAdapter` for the same reasons as discussed above.

125–131 Our handler only overrides one method, `mousePressed()`. This handler begins a new curve.

2.4.2 The Drawing Application

The previous section developed a specialized component, `DrawPanel`. The `DrawPanel` is not a complete program, however. It must be added to a `JFrame` before it can be displayed. No special code is needed. It is just added to the `JFrame` in the same way we previously added buttons and textboxes. As we saw when developing the `DrawPanel`, however, it contains some public functionality like changing drawing colors and clearing the panel, that is only available by calling its methods. Dropdown menus are the ideal way to access these methods.

In this section complete our develop of a drawing program, using a `JFrame` with dropdown menus to display and manipulate a `DrawPanel`.

Swing Menu Classes

There are a number of classes associated with dropdown menus in Swing.

- There is a single `JMenuBar` per `JFrame`. It contains the menus and menu items.
- `JMenu`s are added to the menu bar.
- `JMenuItem`s are added to the menus. `JMenuItems` fire `ActionEvents`.
- Event handlers are registered with the menu items.

The menu for the application consists of

- a single menu, labeled ‘File’
- three menu items in the File menu: ’Color’, ’Clear’ and ’Exit’

```
 1 import javax.swing.*;
 2 import javax.swing.plaf.*;
 3 import java.awt.*;
 4 import java.awt.event.*;
 5 import java.util.*;
 6
 7 /**
 8  * This class implements a drawing program in java
 9  *
10  * @author Stuart Hansen
11  * @version September 2008
12  */
13
14 public class DrawProgram extends JFrame {
15    // These elements form the GUI of the application
16    DrawPanel drawPanel = new DrawPanel();
17    JMenuBar menuBar = new JMenuBar();
18    JMenu menu = new JMenu ("File");
19    JMenuItem colorItem = new JMenuItem("Color");
20    JMenuItem clearItem = new JMenuItem("Clear");
21    JMenuItem exitItem = new JMenuItem ("Exit");
22
23    public DrawProgram () {
24       super ("Line Draw");
25
26       // add the menu to the application Frame
27       setJMenuBar (menuBar);
28       menu.add(colorItem);
29       menu.add(clearItem);
30       menu.add(exitItem);
31       menuBar.add(menu);
32
33       // setup the drawPanel
34       getContentPane().add(drawPanel);
35       drawPanel.setBackground(Color.white);
36
37       // Set the current Color
38       drawPanel.setColor(Color.black);
```

Lines	Commentary
16–21	Declare and instantiate all the components for the application, including the `DrawPanel` and all parts of the menu.
24	The call to `super()` initializes the `JFrame`.
25–30	Setup the menu for the application. We add the menu bar to the application, add the menu items to the menu, and and the menu to the menu bar.
33–38	Add a `DrawPanel` to the application and set a couple of its initial properties.

```
39        // Change the drawing color
40        colorItem.addActionListener (
41           new ActionListener () {
42              public void actionPerformed (ActionEvent e) {
43                 Color oldColor = drawPanel.getColor();
44                 Color newColor = JColorChooser.showDialog(null,
45                                        "Choose a new color", oldColor);
46                 if (newColor != null)
47                    drawPanel.setColor(newColor);
48              }
49           }
50        );
51
52        // Clear the drawing by replacing the DrawingPanel
53        clearItem.addActionListener (
54           new ActionListener () {
55              public void actionPerformed (ActionEvent e) {
56                 drawPanel.reset();
57              }
58           }
59        );
60
61        // Exit the system elegantly
62        exitItem.addActionListener (
63           new ActionListener () {
64              public void actionPerformed (ActionEvent e) {
65                 System.exit(0);
66              }
67           }
68        );
69
70        // Set the application window's properties
71        setSize (400, 400);
72        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
73
74        // display the application window
75        setVisible(true);
76     }
77
78     // a very simple main program
79     public static void main (String args[]) {
80        new DrawProgram();
81     }
82  }
```

40–68 As with buttons, menu items fire `ActionEvent`s. We register an `ActionListener` with each menu item. In this example, we use anonymous inner classes for each of the `ActionListener`s.

40–50 The code within each handler method is relatively short. The handler for the **Color** menu opens a `JColorChooser`. If the user chooses a new color, that color is passed along to the `drawPanel`. If the user cancels, null is returned to the handler, in which case no new color is set.

53–59 The **Clear** handler clears the `drawPanel`'s model. Note that `reset()` contains a call to `repaint()`, so that when the model is cleared, the display is also cleared.

62–69 The handler for **Exit**, exits the application. `System.exit()` takes an integer parameter that encodes why the application terminated. Zero is the standard argument to mean that the application terminated normally.

71–75 The end of the constructor sets some main window parameters for the `JFrame`. The main method starts the program running by creating a new object of type `DrawProgram`.

2.5 List of Doubles

This section presents a complete, longer Java application that maintains a list of floating point numbers, `Double`s in Java. The interface to the application is shown in Figure 2.8.

There are several very simple use cases:

- The user enters a number in the input field and presses `Enter`. The number is added to the list.
- The user may also select an element in the list and delete it by clicking `Delete`.
- The user may clear the entire list by clicking `Clear All`.

Whenever the list is updated, both the `average` and the `maximum` are recalculated and updated, as well. If the list becomes empty, the the `average` and the `maximum` are set to `NaN`, which stands for "Not a Number".

While this is an admittedly contrived application designed for pedagogic purposes, simple statistical applications similar to this one have many uses. Unfortunately for us, a spreadsheet will provide the needed functionality and more.

The example illustrates several of the more advanced points mentioned earlier in this chapter.

- Events are propagated through the system. E.g. pressing enter on the textfield updates the list, which in turn fires events that update the average and maximum. New averages and maximums update the display by again firing events.
- Like most programs, there are many things that can go wrong. We handle exceptions within our event handlers, printing error messages as needed. Figure 2.8 shows the error displayed when a user tries to enter non–numeric data into the list.

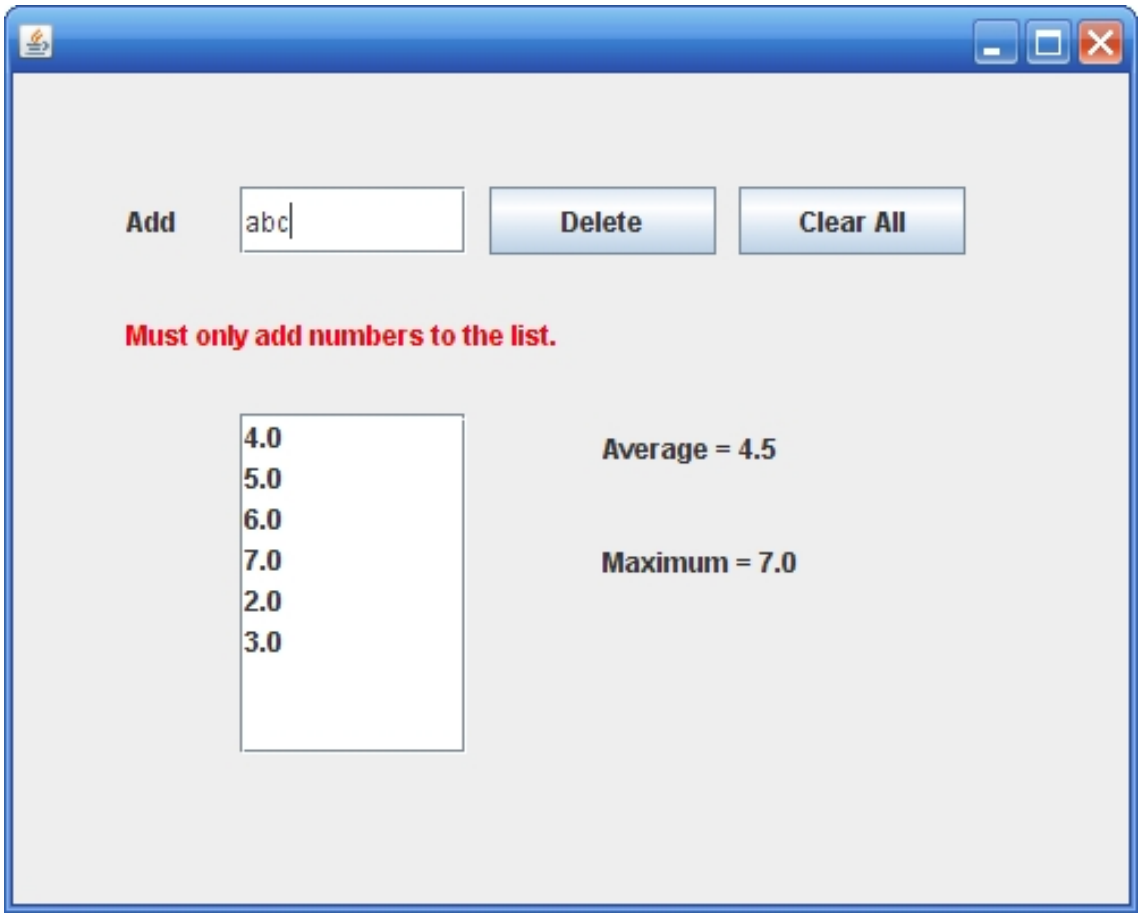

Figure 2.8: The user may add and delete numbers from the list. The application automatically updates the displayed average and maximum using event based techniques.

2.5.1 Double List Model

In the previous example we saw how we could specialize a `JList`'s model. In that example we specialized it so that the list was maintained in lexicographic order.

In this example we also specialize the model, but in a different way. Here, only `Double`s may be added to the list.

```
 1 import javax.swing.DefaultListModel;
 2
 3 /**
 4  * DoubleListModel is the model for the DoubleList class.
 5  * It overrides a couple of methods so that only Doubles are
 6  * placed into it.
 7  *
 8  * @author Stuart Hansen
 9  * @version September 2008
10  */
11
12 public class DoubleListModel extends DefaultListModel {
13    // Overrides add element so that only Doubles are added
14    public void addElement(Object obj) {
15       Double d = Double.parseDouble(obj.toString());
16       addElement(d);
17    }
18
19    // A specialized addElement for Doubles
20    public void addElement(Double d) {
21         super.addElement(d);
22    }
23
24    // Overrides toArray() so that the returned Array contains Doubles
25    public Double[] toArray() {
26       Object[] tempArr = super.toArray();
27       Double[] dArr = new Double[tempArr.length];
28       for (int i=0; i<tempArr.length; i++)
29          dArr[i] = (Double) tempArr[i];
30       return dArr;
31    }
32 }
```

Lines	Commentary
12	The `DoubleListModel` class extends the `DefaultListModel` class overriding several methods in it.
14–17	`addElement(Object obj)` is overridden. The method converts its `Object` parameter first to a `String` and then to a `Double`. In Java all objects have a `toString()` method, so this conversion is guaranteed to succeed. It then proceeds to convert the `String` to a `Double`. This code works correctly if the user has entered a valid `Double`. While not explicit in the code, Java throws a `ClassCastException` occurs if the conversion fails. Finally, we add the newly created `Double` to the list, using the overloaded `addElement(Double d)` method.
20–22	We also implement a specialized `addElement(Double d)` method that adds `Double`s to the list. Note that we are still using the list data structure maintained by the super class, `DefaultListModel`. That list is a list of objects. By controlling access to it by overriding methods, we limit the values that may be added to just `Double`s. `DefaultListModel` contains several other methods that add or modify values in the list. These include: `add()`, `insertElementAt()`, `set()` and `setElementAt()`. A more complete `DoubleListModel` class would also override these methods. Since they aren't necessary for our example, they are ignored here.
25–31	Finally, we override `DefaultListModel`'s `toArray()` method so that the array returned is an array of `Double`s. We will use `toArray()` to get the values for calculating the average and maximum. Overriding `toArray()` keeps us from having to cast each element to a `Double` after accessing it in the original array. As with the previous note, there are several other "getter" methods that could be overridden, including: `elementAt()`, `elements()`, `firstElement()`, `get()`, `getElemetnAt()`, `lastElement()`, and `remove()`. Again, these are ignored, since they are not used by our application.

2.5.2 Average and Max Classes

The `Average` and `Max` classes model the two statistics displayed in the user interface. Whenever the average or the max is recalculated they fire a data changed event that notifies all listeners that they should take appropriate action. Note that from a practical point of view, we could have the average and maximum calculations done by methods within the application, not in separate classes, but that is not in the spirit of demonstrating data changed events.

The developers of Java Beans realized the importance of this type of event and included support for them in the Java Beans package. This support includes:

- **The PropertyChangeEvent Class**
  The `PropertyChangeEvent` class is used to encapsulate the information needed to process property changes. Each instance includes four pieces of information, the event source, the name of the property changed, the original value of the property, and the new value of the property. In java event objects are passive, containing the information needed to handle the events.

- **The PropertyChangeListener Interface**
  All classes wanting to receive `PropertyChangeEvent`s must implement the `PropertyChangeListener` interface. The interface contains a single method, `void propertyChange(PropertyChangeEvent evt)`. Event handling code goes in this method.

- **The PropertyChangeSupport Class**
  There are still two pieces of functionality missing for us to be able to use `PropertyChange` events. We need to be able to register handlers with sources and fire the events to each handler when the event occurs. The `PropertyChangeSupport` class contains numerous methods, but the two that are central to our discussion are: `addPropertyChangeListener()` and `firePropertyChange()`.

To use this class we instantiate an instance of it for each property of interest. We then delegate responsibility for registering listeners and firing events to that instance.

```
 1 import java.beans.*;
 2
 3 /**
 4  *  The Average class finds and keeps track of the average
 5  *  of an array of Doubles.
 6  *
 7  *  @author Stuart Hansen
 8  *  @version September 2008
 9  */
10 public class Average {
11    // We maintain the average in order to have an old
12    // average for the property change event
13    private Double average;
14
15    // We use pcs to facilitate our property change events
16    private PropertyChangeSupport pcs;
17
18    // The default constructor
19    public Average() {
20       average = new Double (Double.NaN);
21       pcs = new PropertyChangeSupport (this);
22    }
23
24    // Return the current Average
25    public Double getAverage() {
26       return average;
27    }
28
29    // Return the current Average as a String
30    public String toString () {
31       return average.toString();
32    }
33
34    // Find the average of an enumeration of Doubles
35    public Double findAverage(Double[] dArr) {
36       Double oldAverage = average;
37
38       if (dArr.length > 0) {
39          double sum = 0.0;
40          int count = 0;
41          for (Double d : dArr) {
42             sum += d;
43             count++;
44          }
45          average = new Double (sum/count);
46       }
47       else
48          average = Double.NaN;
49
50       firePropertyChange (new PropertyChangeEvent (this, "average",
51                                                    oldAverage, average));
52       return average;
53    }
```

Lines	Commentary
1	Import `java.beans.*`. This is the package that contains the `PropertyChange` classes.
16	Declare the `PropertyChangeSupport` object. The `PropertyChangeSupport` class contains methods that allow us to register listeners and fire events to them whenever the `average` is updated.
19–22	The constructor is quite simple. It initializes the average and the `PropertyChangeSupport` object.
35–53	Calculate the average of an array of `Doubles`.
50–51	An interesting part of this method is where it fires the `PropertyChangeEvent`. The `PropertyChangeSupport` class was designed for this purpose. We included the `pcs` object in our class and then use it to register listeners and fire events. The `PropertyChangeEvent` constructor takes four parameters, the event source, `this`; the name of the property changed, in our case `average`; the old value and the new value.

```
55    // Fire a property change
56    public void firePropertyChange (PropertyChangeEvent e) {
57       pcs.firePropertyChange(e);
58    }
59
60    // Add a property change listener
61    public void addPropertyChangeListener(PropertyChangeListener pcl) {
62       pcs.addPropertyChangeListener(pcl);
63    }
64
65    // Delete a property change listener
66    public void removePropertyChangeListener(PropertyChangeListener pcl) {
67       pcs.removePropertyChangeListener(pcl);
68    }
69 }
```

56–68	Place wrapper methods around some of the `PropertyChangeSupport` methods, facilitating public access to them.

The `Max` class directly parallels the code in the `Average` class and is omitted for the sake of brevity.

2.5.3 The Main Class

The `DoubleListMain` class builds the application from the previous classes, Swing components and handlers.

```
 1 import java.awt.*;
 2 import javax.swing.*;
 3 import java.awt.event.*;
 4 import javax.swing.event.*;
 5
 6 /**
 7  * DoubleListMain puts together an application that records numbers
 8  * into a list and reports their average and max.
 9  *
10  * @author Stuart Hansen
11  * @version September 2008
12  */
13
14 public class DoubleListMain extends JFrame {
15    private JScrollPane pane;          // scrollpane for the list
16    private JList list;                // the list's display
17    private DoubleListModel model;     // the model holding the list
18
19    private JLabel add;                // the label for the add field
20    private JTextField inputField;     // where the numbers are entered
21    private JLabel errorLabel;         // a label to display error messages
22
23    private JLabel avgLabel;           // where the average is displayed
24    private Average avg;               // calculates the average
25
26    private JLabel maxLabel;           // where the max is displayed
27    private Max maximum;               // finds the maximum
28
29    private JButton del;               // deletes the selected element
30    private JButton clear;             // clears the entire list
31
32    // The constructor "wires" together the application
33    public DoubleListMain () {
34       Container cPane = getContentPane();
35       cPane.setLayout(null);
36
37       // Set up the list and its model
38       model = new DoubleListModel();
39       list = new JList();
40       list.setModel(model);
41       pane = new JScrollPane(list);
42       pane.setSize(100, 150);
43       pane.setLocation (100, 150);
44       cPane.add(pane);
45
46       // Set up the label for the add field
47       add = new JLabel("Add");
48       add.setLocation(50, 50);
49       add.setSize(40,30);
50       cPane.add(add);
51
52       // Set up the input textField
53       inputField = new JTextField();
54       inputField.setLocation(100, 50);
55       inputField.setSize(100, 30);
56       inputField.addActionListener(new AddHandler());
57       cPane.add(inputField);
```

Lines	Commentary
17, 24, 27	The application's model consists of `model` – which contains the data from the list, `avg` – which contains the average of the data, and `max` – which contains the maximum of the data.
15–30	The GUI consists of the usual assortment of buttons, labels, textfields and lists.
38–44	The model is placed in the list. The list is placed in the scrollpane. The scrollpane is added to the contentPane.
47–50	We add a label and a textfield to the window. We register an `AddHandler` with the textfield as a method of adding to the list. All the handler code appears below.
53–57	We add the textfield used for input to the application.

```
59        // Set up the label for error messages
60        errorLabel = new JLabel("");
61        errorLabel.setForeground(Color.red);
62        errorLabel.setSize(400, 30);
63        errorLabel.setLocation(50, 100);
64        cPane.add(errorLabel);
65
66        // Set up the delete button
67        del = new JButton("Delete");
68        del.setSize (100, 30);
69        del.setLocation(210, 50);
70        del.addActionListener(new DeleteHandler());
71        cPane.add(del);
72
73        // Set up the clear button
74        clear = new JButton("Clear All");
75        clear.setSize (100, 30);
76        clear.setLocation(320, 50);
77        clear.addActionListener(new ClearHandler());
78        cPane.add(clear);
79
80        // Set up the average value and label
81        avg = new Average();
82        avgLabel = new JLabel("Average = NaN");
83        avgLabel.setSize(200, 30);
84        avgLabel.setLocation(260, 150);
85        model.addListDataListener(new AverageAdapter());
86        avg.addPropertyChangeListener(new AvgPropHandler());
87        cPane.add(avgLabel);
88
89        // Set up the maximum value and label
90        maximum = new Max();
91        maxLabel = new JLabel("Maximum = NaN");
92        maxLabel.setSize(200, 30);
93        maxLabel.setLocation(260, 200);
94        model.addListDataListener(new MaximumAdapter());
95        maximum.addPropertyChangeListener(new MaxPropHandler());
96        cPane.add(maxLabel);
97
98        // Set a few windowing parameters and show the frame.
99        setSize(500, 400);
100       setDefaultCloseOperation(EXIT_ON_CLOSE);
101       setVisible(true);
102   }
```

60–64 We place an error message label under the input field. The message is empty, unless an error occurs.

67–78 We add two buttons to the application, one to delete individual items from the list and one to clear the entire list. The registered handlers are defined below.

81–96 We add the `Average` and `Max` objects to the application, along with their GUI representations.

85, 96 Add the `ListDataListeners` to the `model`. Events are fired from the `model` to these handlers when data is added or deleted from the `model`, updating the `average` and the `maximum`.

99–101 We set a few main window parameters and open the window.

2.5.4 Event Handler Classes

All the handlers are defined as inner classes to the application class. This gives them access to all the various application components they need to carry out their activities.

```
104    // This class is the handler for adding numbers
105    private class AddHandler implements ActionListener {
106       public void actionPerformed (ActionEvent e) {
107          try {
108             String str = inputField.getText();
109             model.addElement(str);
110             inputField.setText("");
111             errorLabel.setText("");
112          }
113          catch (Exception ex) {
114             errorLabel.setText("Must only add numbers to the list.");
115          }
116       }
117    }
118
119    // This class is the handler for removing numbers
120    private class DeleteHandler implements ActionListener {
121       public void actionPerformed (ActionEvent e) {
122          try {
123             int index = list.getSelectedIndex();
124             model.remove(index);
125             errorLabel.setText("");
126          }
127          catch (Exception ex) {
128             errorLabel.setText("Error! Use mouse to select element to remove");
129          }
130       }
131    }
132
133    // This class is the handler for clearing the list
134    private class ClearHandler implements ActionListener {
135       public void actionPerformed (ActionEvent e) {
136          model.clear();
137       }
138    }
```

105–117 The `addHandler` is called when the enter key is pressed in the textfield. The data in the textfield is added to the list. Note the `try -- catch` block. As we saw earlier, if the data we try to add, `str`, cannot be parsed into a `Double`, `addElement()` will throw an exception. This exception is caught and an appropriate error message is printed to the `errorLabel`.

120–131 The `DeleteHandler` deletes individual items from the list. We choose the item to delete by clicking on it with the mouse. We then click the *Delete* button, with which this handler is registered. The `getSelectedItem()` method in the `JList` class returns the index of the item that has been clicked. We then remove it from the model. If no item has been clicked an exception is raised, which again is handled by printing a message to the `errorLabel`.

```
140     // This class updates the average
141     private class AverageAdapter implements ListDataListener {
142        public void contentsChanged(ListDataEvent e) {}
143        public void intervalAdded(ListDataEvent e) {
144           Double[] temp = model.toArray();
145           avg.findAverage(temp);
146        }
147
148        public void intervalRemoved(ListDataEvent e) {
149           Double[] temp = model.toArray();
150           avg.findAverage(temp);
151        }
152     }
153
154     // This class updates the maximum
155     private class MaximumAdapter implements ListDataListener {
156        public void contentsChanged(ListDataEvent e) {}
157        public void intervalAdded(ListDataEvent e) {
158           Double [] temp = model.toArray();
159           maximum.findMax(temp);
160        }
161
162        public void intervalRemoved(ListDataEvent e) {
163           Double [] temp = model.toArray();
164           maximum.findMax(temp);
165        }
166     }
167
168     // This handler updates the JLabel when the average changes
169     private class AvgPropHandler implements PropertyChangeListener {
170        public void propertyChange (PropertyChangeEvent e) {
171           avgLabel.setText("Average = " + avg.getAverage());
172        }
173     }
174
175     // This handler updates the JLabel when the maximum changes
176     private class MaxPropHandler implements PropertyChangeListener {
177        public void propertyChange (PropertyChangeEvent e) {
178           maxLabel.setText("Maximum = " + maximum.getMaximum());
179        }
180     }
181
182
183     // A one line main program
184     public static void main (String [] args) {
185        new DoubleListMain();
186     }
187 }
```

134–138 The `ClearHandler` clears the entire list.

141–152 The `AverageAdapter` recalculates the average when data is added or removed from the list.

155–166 The `MaximumAdapter` recalculates the maximum value when data is added or removed from the list.

169–180 The property change handlers update the `JLabel`s in the GUI when the average or the maximum change. The handlers that make the changes to `average` and `maximum` could also set the values in the labels. We use this method to illustrate cascading events, and to demonstrate how to use `PropertyChangeEvent`s.

184–186 The main method starts the application running.

2.5.5 Event Propagation

This example contains a significant amount of code. While all the small pieces of it hopefully make sense, it is worth looking at the big picture, too, to see the cascade of changes that takes place when a number is added to (or deleted from) the list.

To add a number to the list:

1. The number is entered in the `inputField`.
2. `Enter` is pressed which causes the `inputField` to fire an `ActionEvent`.
3. The `AddHandler` runs, updating the `model` with the new value.
4. The `model` fires a `ListDataEvent` which is handled by both the `AverageAdapter` and the `MaximumAdapter`. After updating their internal values, each of these in turn fires a `PropertyChangeEvent`.
5. The `AvgPropHandler` receives the `PropertyChangeEvent` from the `AverageAdapter` and updates the GUI's average label.
6. The `MaxPropHandler` receives the `PropertyChangeEvent` from the `MaximumAdapter` and updates the GUI's maximum label.

A total of four events and five handlers are used to add a single value to the list. An equal number is needed to deleted an element!

2.6 Procedural Event Programming

Throughout this chapter we have discussed event based programming using the language of objects. Our event sources, handlers and application entities were all objects. Event based programming can also be done using a procedural model.

In procedural programming objects don't exist. We have data and *procedures* a.k.a. *functions* that manipulate the data. Functions replace methods. They exist at the top level, not within a class or object. An event handler is a function, typically called a *callback function*. The callback function is still registered dynamically. That is, during a run, not at compile time, we setup what function is called when an event occurs. This is still polymorphism, just based on functions rather than objects.

In languages like C and C++ functions may be passed as parameters to other functions. A function parameter is known as a *function pointer*. GUI libraries like the GL Utility Toolkit (GLUT) contain registration functions that take a function pointer as a parameter and register it to respond to events.

2.7 Summary

In this chapter you have seen several examples of event based programs in Java that have illustrated many of the main features of the paradigm. You are now ready to start developing your own GUI applications. There are dozens of more Swing classes and hundreds of more event classes, but they all work together following the same paradigm. You should be able to read the documentation pages and work with them easily.

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.

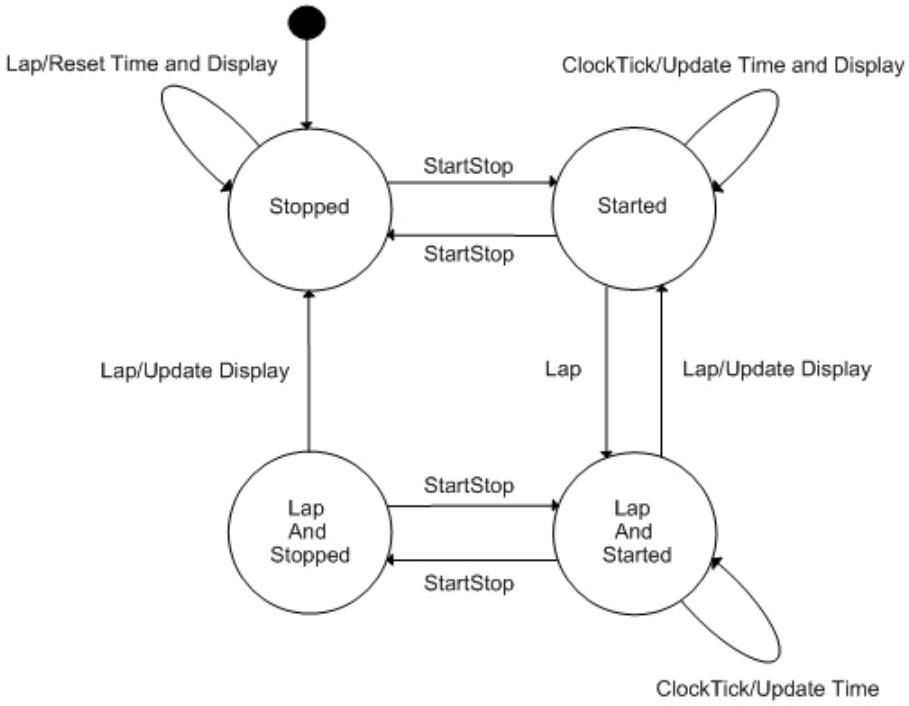

Figure 3.2: The state diagram for the two button stopwatch.

- `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.

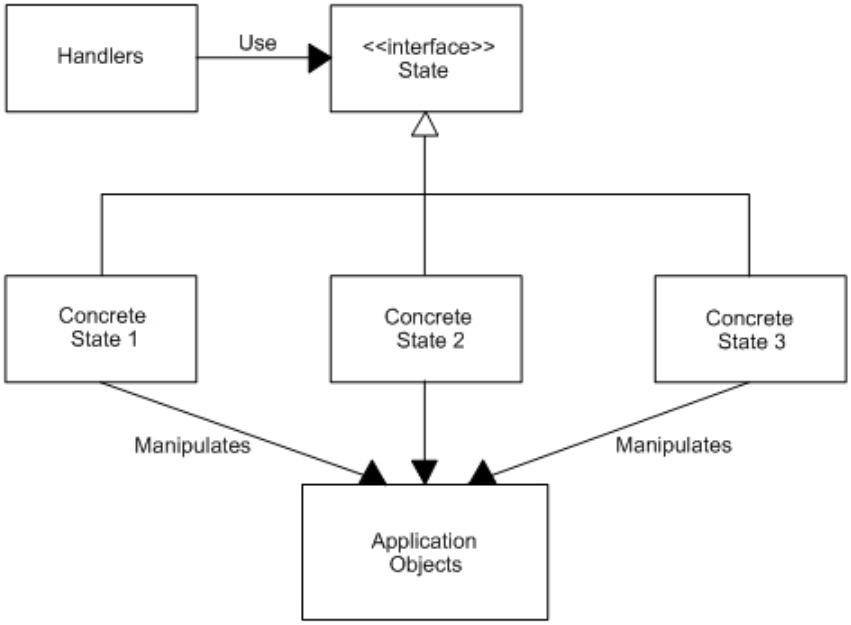

Figure 3.3: The *State Design Pattern* suggests using a different class for each control state present in the system. Handlers invoke methods in the `state` object to manipulate application objects. The handlers do not need to concern themselves with what the control state is, only that it exists.

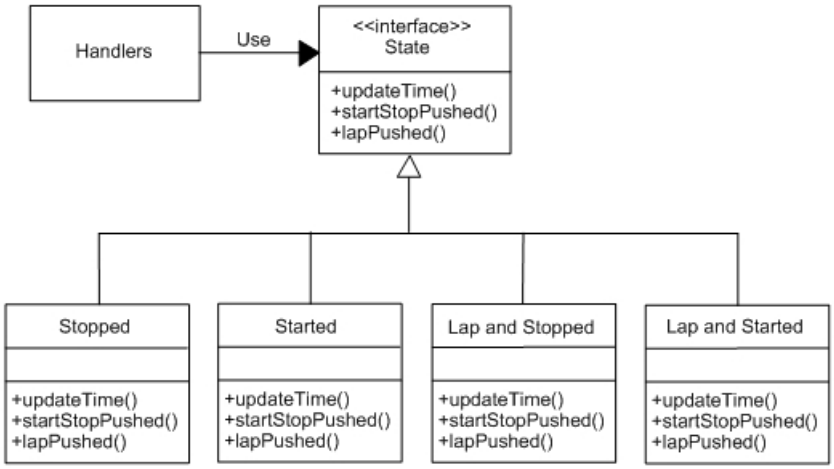

Figure 3.4: The control states for the stopwatch.

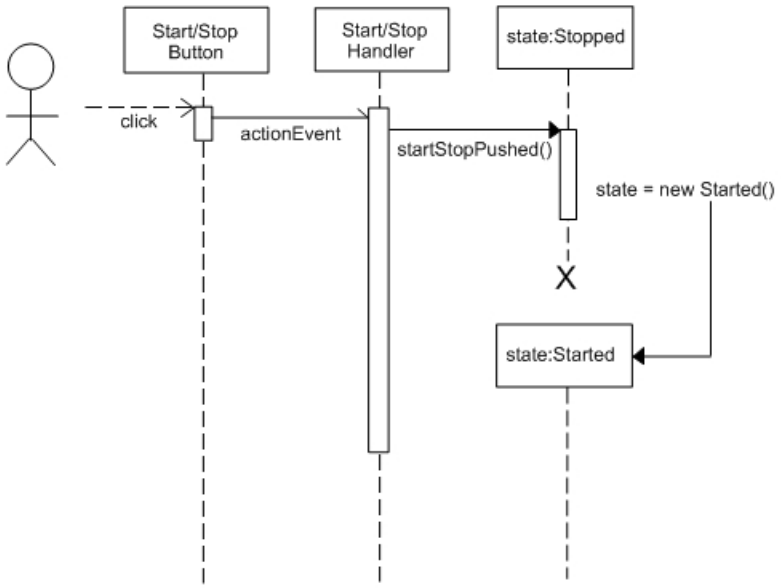

Figure 3.5: This sequence diagram shows the events and method calls made when the Start/Stop button is pushed while the stopwatch is in the Stopped state.

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

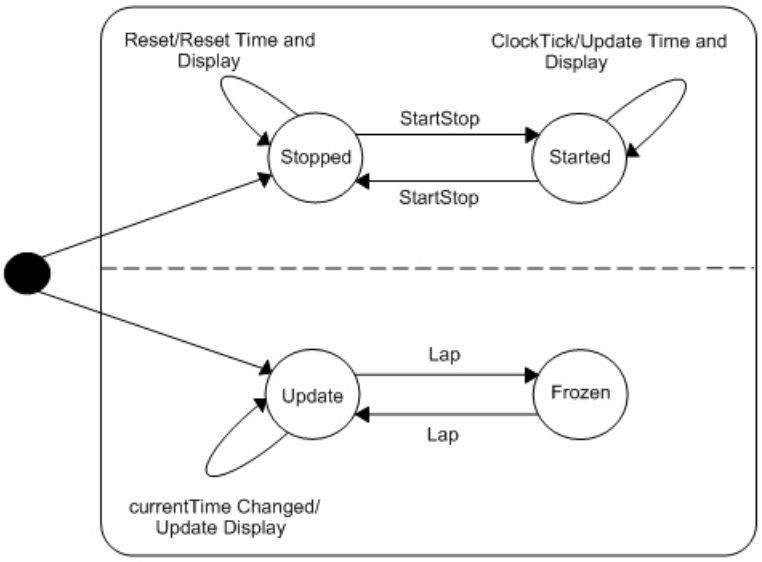

Figure 3.6: The state diagram separates Control and View portions of the stopwatch. Control contains two substates, Stopped and Started. The view also contains two substates, Update and Frozen.

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.

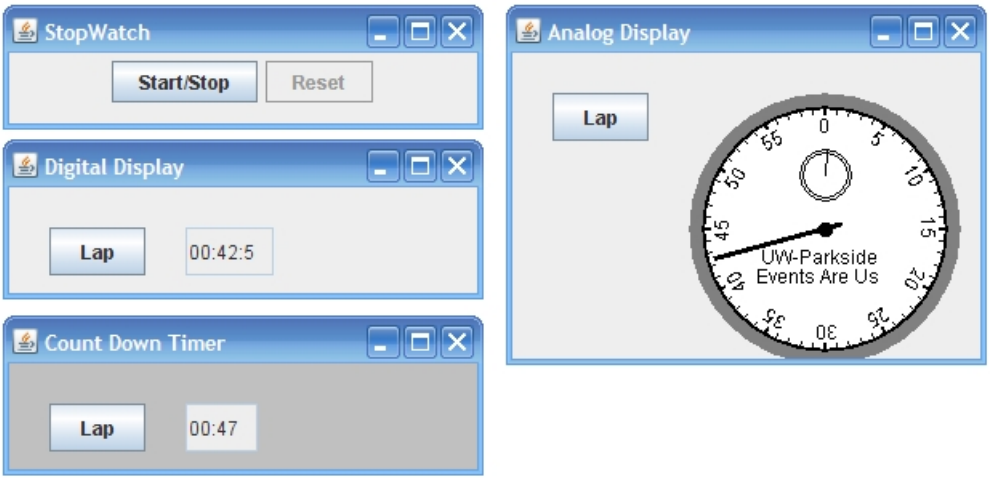

Figure 3.7: The control for the stopwatch is shown in the frame with the Start/Stop and Reset buttons. Each of the other frames shows a distinct view of the stopwatch. The Analog and Digital displays show the same time, represented in very different formats. The countdown displays the amount of time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

Chapter 4

Event Infrastructure

4.1 Introduction

Hopefully we have convinced you in the first several chapters that event based programming is a distinct paradigm from algorithmic programming (either procedural or object–oriented). An event firing has different semantics than a method call. Its unique semantics require additional support from the underlying system. The support infrastructure may be implemented in hardware or software, but must be there.

Languages and libraries that support event based programming include an infrastructure of event services that makes developing and running event based programs easier. Programmers frequently write code that use these services without really understanding their details, treating event processing as a black box. Their code compiles and runs, using the event infrastructure, but the programmer never really knows how the pieces fit together to get the job done.

Our purpose in this chapter is to open up the black box and look in detail at the various features that support event processing. The details obviously depend on the language, library or hardware used. However, there is a common core of services needed in any event based infrastructure. Understanding the ideas behind the core services helps us gain a deeper understanding of how event based systems work. This is necessary if we are going to do more complex event based programming, like designing our own event classes.

Event based programming is distinct from algorithmic programming. Algorithm design deals with how to represent and process the data necessary to carry out a computation and how to decompose the algorithm into simpler, more manageable parts. Algorithm decomposition is often accomplished through the use of procedures (or subroutines) that carry out sub-tasks of the algorithm. The algorithm is accomplished by carrying out processing steps in the procedures, each of which contributes to achieving the goals of the algorithm.

How are event based systems different from algorithms? First, an event based system is a *system* that has parts that interact with each other and that has behaviors that change over time. The parts of an event based system are typically heterogeneous, as well as spatially and temporally separated. The relationships among the parts are dynamic. Moreover, event based systems interact with the external environment, making their inputs – and therefore their behaviors – unpredictable.

As we have already described in section 1.4.2, event based systems are comprised of loosely coupled parts that interact with each other via events. In a procedural setting, when algorithm A calls procedure P, control is passed from A to P, and P maintains control until it returns to A. In an event based setting, when an event E is fired in the context of an event source S, processing in S can continue to take place in parallel with the handling of the event E.

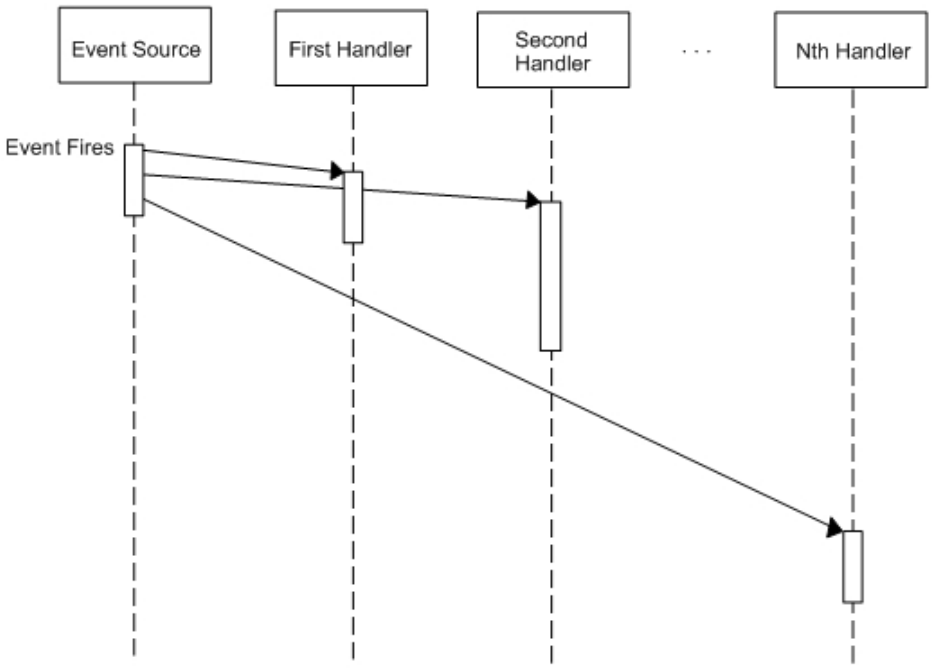

Figure 4.1: This figure shows the critical times during an event's life, when the event fires, when event handling begins and when the last handler completes its processing.

Finally, as an event propagates through a system, it may *morph* or change. "Low-level" events morph into "high-level" events taking on meaning within their current context. For example, a mouse click at the hardware level may be represented by a bit that is set in a device controller, but as the click is processed it takes on successively more characteristics such as screen coordinates, being identified with a process, and finally being regarded as activating a menu item.

As we saw in the previous chapter, event handlers are programmed in much the same way we program procedures and methods. As we have just described, however, the semantics of events are quite different than procedure calls. The event processing infrastructure exists to support the event semantics.

4.2 The Event Life Cycle

Naively, we have described events as being fired and handled almost simultaneously. That is, when an event such as a mouse click fires, we often perceive no delay between the firing of the event and the handling of the event. In truth, when an event occurs, a possibly complex sequence of steps is performed that ultimately ends up with the event being processed. Firing an event includes creating a data structure or object that encapsulates the critical event data and storing it in a place accessible to the event dispatching infrastructure. Sometime thereafter, the event handlers begin executing, and still later, with possible interruptions in-between, the last handler for the event finishes. These steps are illustrated in Figure 4.1. If many events occur in rapid succession, there may be a noticeable delay between when an event fires and when handling begins.

4.2.1 Event Handling Requirements

There is one universal requirement for event based systems:

> Sometime after an event fires, the system achieves a state consistent with the behavioral requirements defined for the event.

For example, in our drawing program in Chapter 2, we pressed and dragged the mouse and we expected a line to be drawn. If the line wasn't drawn, the system wasn't functioning correctly. *Behavioral requirements* tell us how a system is supposed to function. Every event has behavioral requirements, possibly varying based on the control state of the system.

Note that there is wiggle room in this definition. Achieving "a state consistent with the behavioral requirements" is not the same as saying that the handlers for this event executed in a particular order. Reordering or overlapping event handler execution and possibly combining handlers are both acceptable, as long as the system achieves a consistent state.

4.2.2 Timing Requirements

In event based systems, the requirements may include deadlines for completing event processing. As we saw in Chapter 1, event based programming is used to implement many types of systems. To a certain extent, the type of system guides the requirements. For instance, real–time and embedded systems frequently have hard deadlines for their event handlers to complete.

- Consider driving down the highway with your cruise control system on. A deer leaps out in front of you and you slam on the brakes. You want the cruise control system to disengage within a few milliseconds after touching the brakes, not one second or even a half second later. There is a very important deadline for handling the braking event that must be met for the system to work correctly.

- By contrast, when you press the down button to summon an elevator, the button should be backlit almost instantly, and you expect the elevator to stop at your floor – but if you wait one or two minutes for the elevator to arrive, it is not likely to matter.

If systems have timing requirements we should recognize those requirements explicitly and design our systems to guarantee that they are met.

4.2.3 Concurrent Event Handling

The final piece of background we need before getting into the infrastructure details is an understanding of *concurrency*. If two things are happen in overlapping time spans, we say they happen *concurrently*. For example, in real–time systems it is common to find multiple dedicated processors handling events. The events are handled concurrently making it easier to meet the required deadlines.

Similarly, many modern desktop computers have with multiple *cores*. You can think of cores as multiple processors, all physically located on one chip. Software doesn't automatically use more than one core. Programmers must design and implement their systems to take advantage of the additional computing power.

Modern languages like Java use an abstraction called a *thread* to implement concurrency within a program. For example, GUIs often use a separate thread for event handling. The thread loops forever, repeatedly looking for the next event and invoking its handlers. This topic is discussed more completely in the next chapter.

4.3 Event Infrastructure Services

There are two core event services that any event infrastructure should supply: *registering/unregistering handlers* and *event dispatching*. There are other event services that may be provided, too. For example, logging events can be a great help in tracking down problems, and support for remote events can help the programmer develop distributed systems. These advanced services are not covered in this chapter.

4.3.1 Event Registration

As we have seen, an event handler registers its interest in a particular event with the event's source. The event source keeps track of the handler until either it is unregistered or the application terminates.

Some event systems use *unicast* events. Unicast event sources have at most one handler for each event. For example, *The GL Utility Toolkit, (GLUT)* which is part of *OpenGL* uses unicast callback functions for handlers. In GLUT there is one callback function to handle an application's mouse events. Registration of unicast event handlers is simple: The event source contains a reference to the handler.

Multicast event systems may have multiple handlers for each event. When an event fires each of the handlers is notified. This may sound confusing, but multicasting is not as unusual or complicated as it may first seem. For example, consider our double list example from the previous chapter. There were three handlers registered to receive the data changed events: the list's view, the average calculator and the maximum calculator. Each had clearly defined responsibilities. All three were notified when the event occurred.

Supporting multicasting requires event sources to maintain a list (or other data structure) of handlers. It also makes event dispatching a bit more complicated.

4.3.2 Event Dispatching

The *event dispatcher* is responsible for invoking the event handlers when an event fires[1]. Broadly speaking, we divide event dispatching approaches into two categories: *push* and *pull*.

- In *push* dispatching, the event source is responsible for activating the dispatcher when an event occurs. We say that the event source pushes the event to the dispatcher. The dispatcher is then responsible for invoking the handlers. Note that it is possible to have the event source call the event handlers directly. In this case the source is also the dispatcher.

- In *pull* dispatching, the dispatcher periodically queries or *polls* the event sources for events. When it finds one, it calls the relevant handlers.

The difference between *push* and *pull* dispatching is really one of active responsibility. In a pull system, the dispatcher is responsible for detecting that an event has occurred, typically by polling the event source. In a push system, the event source activates the dispatcher or enqueues the event to be handled later. A few examples may help clarify the differences.

[1] In some sense the event dispatcher is itself an event handler. It is code that executes as a result of an event occurring. For purposes of our discussion, however, we distinguish between code that is supplied by the infrastructure, the dispatcher, and code that is application specific, the handlers.

Push and Pull Examples

Suppose that you work for XYZ company as a sales representative. Your objective is to take orders from customers and to process these orders. In this example, receiving an order from a customer is the particular event we are interested in.

As a sales representative, you might simply sit at your desk waiting for customers to call you to place orders. You register your willingness to accept orders by giving your customers your office phone number. This is an example of push dispatching, since the customer (the event source) is calling you (and thereby activating you, the event dispatcher) to process the order (handle the event). You can sleep at your desk until your phone rings.

On the other hand, you might have a set of customers that you visit periodically on a sales route. As you visit each customer, you ask if they have any orders to place (event polling), and you process any orders that they may have. It may well be that a particular customer has no orders to place during your visit, in which case you will simply return on your next visit to check again.

It's possible, of course, for XYZ company to combine these two approaches – the company may well want to do so to maximize their opportunities to receive orders by phone as well as to pay personal attention to their best customers. However, in event based computing systems it is rare to have events dispatched with both push and pull approaches.

Systems Using Pull Dispatching

Operating systems are responsible for interacting with peripheral devices such as a mouse, keyboard, or disk drive. For example, when you click a mouse button, the hardware device controller for the mouse (typically a chip on a computer motherboard, but not part of the main processor) may register the fact that the button has been clicked by setting a bit in an internal device controller register. The operating system can periodically check the device bit to see if it is set, and if so, the operating system can treat it as a mouse click event. The device controller will normally reset the bit after it has been read, so that the device can register multiple mouse click events.

Both the **X11 Window System** and **Microsoft Windows** generate myriad events such as mouse clicks (at a higher level of abstraction than described above) and window expose events. In an application, these events are typically processed in an *event loop* that might look something like this (*WARNING:* `C` *code alert*):

```c
/* Xlib event loop */
while (1) {
  XNextEvent(display, &report);
  switch (report.type) {
    case Expose:
      /* handle an Expose event ... */
      break;
    case ConfigureNotify:
      /* handle a ConfigureNotify event ... */
      break;
    case ButtonPress:
      /* handle a ButtonPress event ... */
      break;
    ...
    default:
      break;
  } /* end switch */
} /* end while */
```

The loop repeatedly requests the next `X11` event and, based on the event type, handles the event appropriately. The event loop also elucidates why we use the term *dispatcher*. It receives events of many different types and, based on the type of an event, dispatches each to the appropriate handler.

The graphics packages `OpenGL` and `GLUT` similarly use event loops to handle windowing events [Angel, 2008, Shreiner et al., 2007]. After registering various event handlers, the `GLUT` event loop `glutMainLoop()` is called, which serves the same purpose as the `Xlib` event loop given above.

Systems Using Push Dispatching

- In our example using pull dispatching, we described how an operating system could poll a mouse device controller bit to determine when a mouse button has been pressed. Most high-performance operating systems do not use polling, however, because of the overhead associated with it. Instead, high-performance systems (including most consumer-oriented workstations and laptops) use *interrupt processing* to handle such events.

  An interrupt-driven device has the capability of notifying the computer processor that an event (such as a mouse press) has occurred. When an interrupt occurs, the processor temporarily suspends the current program and identifies what device caused the interrupt by probing external bus lines. The processor then looks up the address of the device interrupt service routine (found at a fixed location in memory based on the device identification) and executes the code found at that address. When the interrupt service routine has finished, the processor resumes the suspended program. In this case, the interrupt service route plays the role of the event handler.

  Notice that for interrupts to work, the address of the interrupt service routine must be installed at the proper location in memory corresponding to the device. This is how the handler gets registered with the event source.

- Java and Microsoft .NET both use push dispatching. You have already seen examples in Chapter 2 that illustrate how handlers are registered with event sources in Java.

The Event Queue

In push dispatching, the event dispatcher may activate an event handler by calling it directly. If the handler does not return quickly enough, a second event may arrive at the dispatcher while the first event is still being handled. Two potentially bad things could happen in this case: either the dispatcher could try to call the event handler again – with possibly unpredictable results [Stallings, 2008, Tanenbaum, 2007], or the dispatcher could discard the second event – again with unpredictable results.

A good way to resolve this situation is to store the second event until the first one completes its processing. What if multiple events occur while the first is being handled? Each of the events should be stored until processed. In many systems, a dedicated *event queue* exists to handle this situation. The event queue is a data structure holding events waiting to be handled. When the event dispatcher finishes processing one event it gets the next event from the queue.

An event queue must have a large enough capacity to store pending events without the likelihood of running out of queue space. If the event handler takes too much time to finish, events can pile up in the queue so that the queue fills to capacity. In this case, subsequent events may be discarded, or the system may crash – neither of which is desirable. Good event based system design will have handlers that return quickly and event queues that are sufficiently large to handle pending events. Obviously ,such designs must take into account processor and device speed and must carefully craft the handlers to optimize their performance.

As mentioned above, it is possible for each event source to function as its own event dispatcher; having its own event queue, but this would be a waste of system resources. Each event queue would need to have a capacity large enough to accommodate the worst case number of pending events. Instead, most systems have a single event dispatcher with a single event queue that holds all pending events. Each event source is responsible for enqueuing (pushing) its events. The dispatcher, running in a separate system thread, dequeues the events and activates their respective handlers.

Event Ordering

Events are generally put on the event queue in the time sequence order in which they were fired, so that if event A occurs before event B, event A will be put on the event queue and handled before event B. This behavior is called *event ordering*. Observe that event ordering is another reason to have a single event queue rather than one for each event source: having multiple queues would mean that event ordering might be preserved by each queue, but there would be no guarantee that event ordering would be preserved between queues.

Even with multiple queues, event ordering can be preserved if the queued events have *timestamps* that record the time at which they were fired. The dispatching mechanism would then be expected to handle events in the proper order based on their timestamps. The situation becomes more complicated in a distributed system: one in which the event sources are on different hardware platforms and perhaps are physically distant from one another. In this case, timestamps would be necessary to ensure that event ordering was preserved in the presence of network delays. But this also requires that all the entities in a distributed system would need to share exactly the same system time – a problem in its own right [Lamport, 1978].

Event Coalescing

An event dispatcher may also implement *event coalescing*. Coalescing events is the process of combining multiple events into one. Sometimes nearly identical events occur such that handling the second event separately would leave the system in the same state as having the handler called only once. Also, duplicate events can occur in rapid succession because of physical phenomena. Events waiting in the queue can be examined, and if there are events that can be safely coalesced, the system does so.

A practical example of event coalescing is an elevator interface. A hotel guest requests elevator service at a particular floor by pressing the down button. If more than one guest is waiting before the elevator arrives, or if a guest grows impatient, the button may be pressed multiple times, but there is no need for a downward traveling elevator to service the floor more than once. The button presses are coalesced into a single event.

Another example is window repainting. Some graphics systems use events to signal that a display window (or region) needs repainting. If a program tells the system repeatedly that a window needs repainting, there is no need to repaint more than once, as after one repainting, the window will be up-to-date[2].

A final example is keyboard debouncing. A human operating a keyboard or keypad is able to press keys at a maximum rate of less than 20 per second [Norris McWhirter, 1985], giving a minimum 50ms delay between key presses. Sometimes keyboard devices, because of physical or electrical properties, may record two key presses within a short time frame (1ms to 2ms) that cannot possibly be the result of user interaction with the device. Such a phenomenon is called *keybounce* [Hyde, 2003]. To counteract keybounce, two key press events from the same human-operated device should be coalesced into one if they occur within a few milliseconds of each other (based on

[2]In modern versions of Java, repaint coalescing is handled by the `RepaintManager`, not by the event dispatcher.

their timestamps, for example). The failure to do so can have serious unintended consequences [Food and Drug Adminstration, 2006][3].

4.3.3 Hybrid Dispatching

The pull-push differentiation provides a nice conceptual model and does a good job describing many dispatching systems. More complex models may be required when three or more active agents are involved.

In distributed event systems, the source, the event dispatcher, and the handler(s) may exist on separate computers. The source may push an event onto the dispatcher's queue, or the dispatcher may poll the source. Similarly, the dispatcher may push an event to the handlers, or handlers may poll the dispatcher's queue to see if there are any events. We will discuss these issues more completely in a later chapter.

Publish-subscribe systems have a similar three-tier structure [Muhl et al., 1998]. Publishers deliver (push) content to a repository agent. Subscribers either register their interest in published content with the repository agent which then delivers (push) the content to the subscribers, or the subscribers scan the repository (pull) for content that they want to receive.

4.4 Java's Support for Events

To illustrate the practical issues of implementing an event infrastructure, we look at how it is done in Java.

4.4.1 Handler Registration

Multicast events require the event source to maintain a list of handlers. Java Swing components can multicast their events, so all Swing components, e.g. `JButton`s, `JTextfield`s, `JPanel`s, etc., maintain a list of listeners. All these components inherit (directly or indirectly) from `JComponent`, so it makes sense to declare the list there and use inheritance to obtain access. `JComponent` contains the declaration:

```
1    protected EventListenerList listenerList = new EventListenerList();
```

`listenerList` is the name of the list. The `EventListenerList` class is a wrapper around an array of `Object`s. This give the Java virtual machine fast access to all the listeners.

To add handlers to the list, we invoke an `add***Listener()` method. For example, `JButton` inherits its `addActionListener()` method from its parent class, `AbstractButton`. The `addActionListener()` code is:

```
1 public void addActionListener(ActionListener listen) {
2     listenerList.add(ActionListener.class, listen);
3 }
```

Note that on line 2, `listenerList`'s `add()` method takes two parameters. This requires a bit of explanation. As you should recall from Chapter 2, Swing components can fire all sorts of events, e.g. `ActionEvent`s, `MouseEvent`s, `MouseMotionEvent`s, `KeyEvent`s, etc. The `listenerList` is used for all of these, not just `ActionEvent`s. When an event fires, the component needs to search the list for just the appropriate listeners. The code above adds both the class (`ActionListener.class`) and the handler (`listen`), to speed up searching the list. An argument can be made that multiple

[3]Note that keybounces can also be caused by the user and can be a serious impediment to typing for individuals suffering from tremors [Trewin and Pain, 1998].

lists of handlers, one for each type of event, should be maintained. While this would alleviate the need for searching the list, there would be significant overhead within each component, much of it never used.

4.4.2 JCL's Event Classes

Java's class libraries come with hundreds of event classes. The classes form a subtree within Java's class hierarchy: see Figure 4.2. At the root of the subtree is `EventObject`. `EventObject` is a very simple class. Since all Java events contain references to their source, there is a protected instance variable named `source`, as well as a public `getSource()` method.

The event tree's complexity is derived in part from Java evolving over time. Initially, Java just included the *Abstract Windows Toolkit (AWT)* for GUI development. `AWTEvent` is a direct subclass of `EventObject`. All events that work with the AWT GUI classes inherit from `AWTEvent`. When Swing was added to Java, newer Swing event classes were added into the hierarchy. Many were added as subclasses to `AWTEvent`. To make matters a bit more confusing, other Swing event classes inherit directly from `EventObject`.

AWT Event Classes

All our Java GUI programs used events that inherited from Java's `AWTEvent` class.

`AWTEvent`s augment `EventObject`'s data with an integer event ID that is used to speed searching the `listenerList` for events of a particular type, as well as numerous static constants and specialized fields.

There are many subclasses to `AWTEvent`. These classes store the information needed to describe the specific type of event. The instance methods are primarily "getters" for the values. Because events may be processed by multiple handlers, it is a bad idea to change event attributes during processing. Therefore, most event objects are *immutable*. They do not include "setters" that would allow us to change attributes.

Take, for example, `MouseEvent`. `MouseEvent`s occur when a button is pressed, released, or clicked; when a component was entered or exited; or when the mouse wheel was moved. The `MouseEvent` class implements the following methods:

- `getButton()` which returns an integer telling which, if any, button was pressed.
- `getX()` which returns the X location (in the source's coordinate system) where the event occurred.
- `getY()` which returns the Y location (in the source's coordinate system) where the event occurred.
- Several other getters used in various situations.

4.4.3 Java Event Processing

The steps Java takes to process an event depends on the event's location in the class hierarchy.

Dispatching AWTEvents

Java's virtual machine uses a separate thread, known as the *event-dispatching thread*, to process `AWTEvent`s. In GUI programs the thread is typically started when `setVisible(true)` is called, but the thread may also be started in a number of other ways. All `AWTEvent`s are processed by this

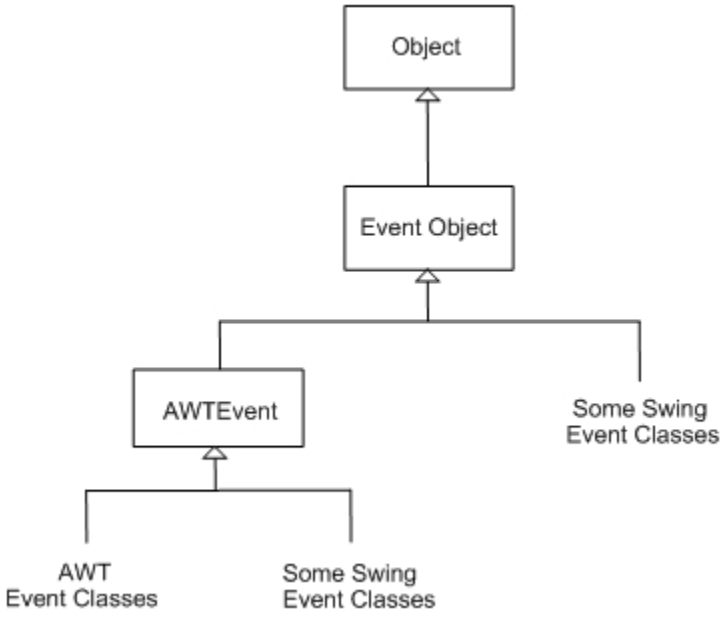

Figure 4.2: Overview of the Java event class hierarchy.

thread. The event-dispatching thread handles events in a first-fired first handled order, see Figure 4.3.

Thread Safe Components

As discussed earlier, if a handler takes too much time, the GUI will appear unresponsive and the programmer should consider using a separate thread for that particular handler.

By contrast, some handler code may be required to execute only in the event-dispatching thread, such as handlers that access GUI components. For example, a textfield may have its text reset by a handler, or a component may have its background color changed. These types of activities need to be done within the event-dispatching thread, as Java's Swing classes were designed using the *single thread rule*. Simply stated, once a Swing component is visible, only the event-dispatching thread should access it. A few of a Swing component's methods are considered *thread safe*, like adding and removing listeners and `repaint()`, and are exempt from the rule. All other component method calls should follow the single thread rule.

This leaves us with a bit of an awkward situation: If you have a long running handler, it should execute in its own thread, but if it needs to access its source component, it should run in the event-dispatching thread. Java provides a work around. The programmer can specify that code be executed in the event-dispatching thread using the `SwingUtilities` class methods `invokeLater()` and `invokeAndWait()`.

Processing AWT Events

When an `AWTEvent` fires, the source creates an event object and enqueues it in the system event queue. The event queue is an instance of `EventQueue` and exists as an object in the Java virtual machine. It is possible to obtain a reference to the queue by calling the default `Toolkit` object's

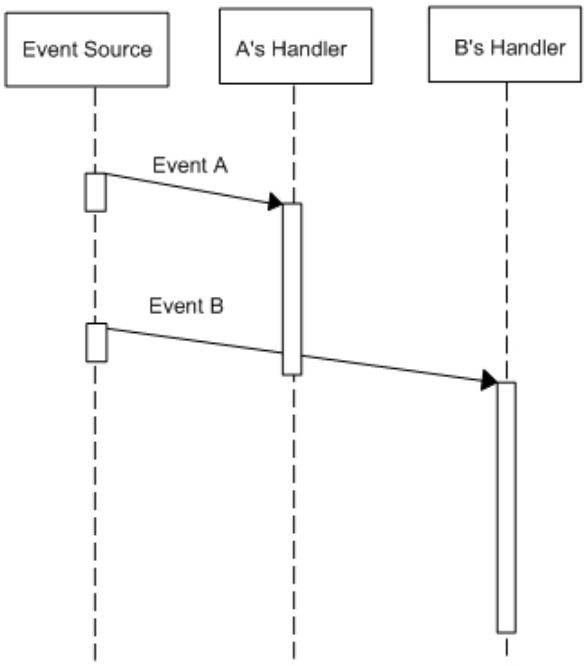

Figure 4.3: Java's AWT event model requires that events are handled sequentially, in the order they are fired. Thus, *Event A*'s handling completes before *Event B*'s handling begins.

`getSystemEventQueue()` method. The reference is obviously necessary if we want to explicitly place events into the queue. It is also possible to replace the default event queue with one of your own devising using the event queue's `push()` method. This is useful if you want to instrument the event queue to log the events as they take place, i.e. for debugging or testing purposes, or if you want to change the semantics of the event queue, such as implementing a priority hierarchy of events for a particular application.

The event-dispatching thread is responsible for processing `AWTEvent`s after they arrive in the queue. The processing includes:

1. Get the next event from the queue by calling the queue's `getNextEvent()` method.

2. Pass the event along as an argument to the queue's `dispatchEvent()` method. `dispatchEvent()` behaves as follows:

   (a) If the event implements the `ActiveEvent` interface, it calls the event's `dispatch()` method. Java uses the `ActiveEvent` interface for events that know how to process themselves. As we will see in the last section of the chapter, `ActiveEvent`s are a good way to develop our own event classes.

   (b) Otherwise, if the event source is a `Component`, it calls the `Component`'s `dispatchEvent()` method, again passing along the event as a parameter. Note that Swing components inherit from `JComponent` which in turn inherits from AWT's `Component`, so this is the path followed when processing most GUI events. `Component`'s `dispatchEvent()` runs through a variety of cases, handling both *low-level* and *high-level* events.

Low-Level and High-Level Events

Low-level events include mouse events, keyboard events and windowing events, like resizing the window. These events are created at the operating system level and passed into our application for handling.

By contrast, *high-level* events are those that aren't tied to any particular device. For example, `ActionEvent`s are high level, as they may be fired by a mouse click, or a key press, choosing a menu item. You may think of a high-level event as more loosely coupled to hardware and more tightly coupled to the application than low-level events.

Some low-level and high-level event types are paired. For example, clicking the left mouse button may be both a `MouseEvent` (a low-level event) and an `ActionEvent` (a high-level event). In Java, if both a high-level handler and a low-level handler are registered for an event, the low-level handler is ignored.

Since there is a small fixed number of low-level event types, `dispatchEvent()` handles them directly.

For high-level events, the source's `dispatchEvent()` calls its `processEvent()` method, again passing along the event object. `processEvent()` checks if any event handlers have been registered for this type of event. If so, it calls each handler in turn.

Method Calls vs. Events Revisited

For many events, `processEvent()` could have been called directly by the source when the event fired, but wasn't. Instead, the event is placed in the event queue and the event dispatching thread ends up invoking `processEvent()`. We gain two things by having `processEvent()` called via this more indirect route. First, the processing is now done asynchronously in the event-dispatching thread, freeing up the GUI thread for other work; and second, events are processed sequentially, in the order they are fired. Both of these provide further decoupling between the event source and its handlers.

We also note that there is only one `processEvent()` method in any component, even if the component fires many different types of events. It is up to `processEvent()` to sort out what type of event has fired and process it appropriately. `Component`'s `processEvent()` may be overridden, but one should be *VERY* careful doing so, to guarantee that all events are processed appropriately.

(c) Otherwise, handle AWT menu events. Note that Swing does not share these menu classes.

(d) Otherwise, ignore the event.

3. Repeat the process for the next event.

Processing Beans-Like Events

The event processing discussed in the previous section works well. It was introduced in Java Version 1.1 and has changed little since. There are a couple of drawbacks to this approach, however.

First, developing *test harnesses* for the events is difficult. A test harness is a collection of code that tests the various parts of our system. Testing should not be done haphazardly, but should be repeatable. A test harness should be able to automatically reproduce the same sequence of events each time the test is run. Unfortunately, with a few exceptions, it is not possible to explicitly fire `AWTEvents` from within code. The firing of events – creating and enqueuing of the event objects – is done by the event sources, but the sources do not contain public methods that allow the tester to recreate the process. For example, there is no `fireMouseClickedEvent()` method available. The

easiest way to test the code is to physically click the mouse. This problem was partially addressed by the introduction of the `Robot` class in Java 1.3. The `Robot` class was specifically designed to generate low-level events and pass them on to the application for testing purposes.

A second problem is that it takes detailed knowledge of the event queue and the `AWTEvent` class to create new `AWTEvent` subclasses. For example, developing new event sources requires the programmer to obtain a reference to the event queue and add events to it. She must also provide processing methods that handle all event types that the new sources may fire.

Event classes that inherit directly from `EventObject` avoid these problems. They are are coupled with event sources that process the events directly, avoiding the need for the event queue. This is the approach taken by Java Beans, Java's component architecture [Englander, 1997]. Each event source contains `fire***Event()` for each type of event it may fire. For example, `JMenu`s work with `MenuEvent`s. `MenuEvent`s include such things as selecting a menu, and canceling a menu[4]. The `MenuEvent` class inherits from `EventObject`, so the `JMenu` class contains the following methods:

- `fireMenuSelected()`
- `fireMenuDeselected()`, and
- `fireMenuCancelled()`

Each of these methods creates an immutable `MenuEvent` object, and passes that object to each of the registered `MenuListener`s. When processing the list of listeners, the list must be cloned, so that if a handler adds or removes listeners, it will not affect which handlers are called for this event, only future events.

Non-`AWTEvent`s are handled within the thread where they originate. This means that cascading events (events firing other events) are processed in a way analogous to procedure calls. If an event handler fires a second event, the second event is processed to completion before the thread of execution returns control to the original handler, see Figure 4.4.

4.5 Programmer Defined AWT Events

One way to get a better grasp on all the ideas in this chapter is to look at an example. In this section we will develop an example using the `AWTEvent` classes. In the next section we will contrast this implementation with a second one developed using a more beans like approach.

Only code snippets are presented here, but the entire listing is available on the text's web site.

> *Patients in a hospital may be hooked up to numerous different types of monitors. The output from these monitors is displayed at the nursing station at the end of the hall. This allows one or two nurses to care for an entire wing of patients. If everything remains within normal ranges, a display shows the monitor's values. If some value exceeds a threshold, warnings are generated, alerting the nurses that a patient needs attention.*
>
> *As the heart pumps, a patient's blood pressure goes up and down with each beat. The higher pressure is known as the* systolic pressure*. The lower pressure is the* diastolic pressure*. Normal values are 120 and 80 millimeters of mercury, respectively. In this example, we will design a system that will sound a warning if the systolic pressure exceeds 140 or goes below 90.*

Figures 4.5 and 4.6 show the simulated patient GUI and monitor windows, respectively.

[4]Note that selecting menu items fires `ActionEvent`s, not `MenuEvent`s.

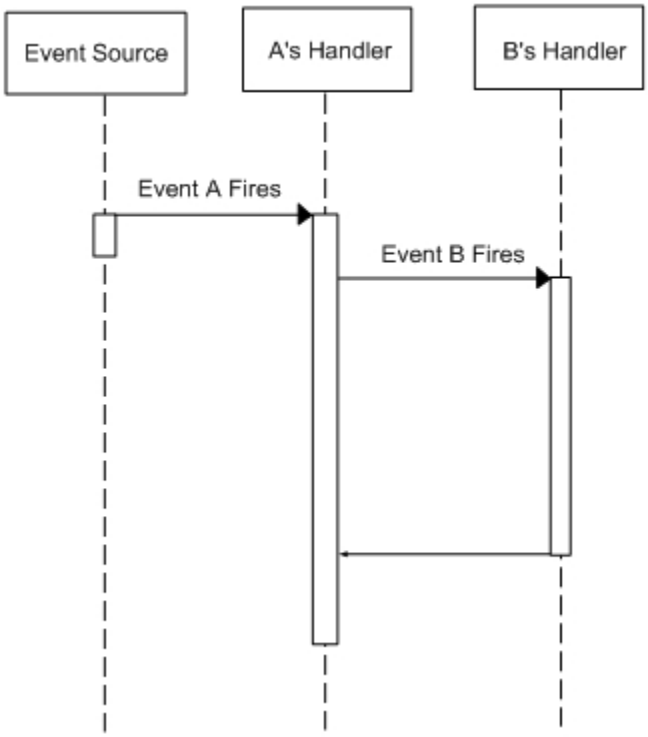

Figure 4.4: Java events that do not inherit from `AWTEvent` are handled in a depth first fashion, much like procedure calls.

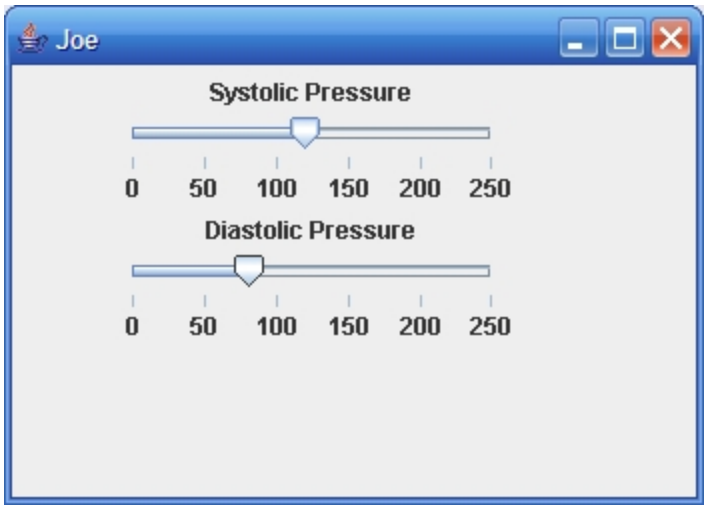

Figure 4.5: The GUI interface for a simulated patient. The patient's blood pressure values are changed by moving the sliders.

There are really three different, but closely related events being used in this example. Any change to the systolic or diastolic pressure will fire an event that updates the display at the nurse's station. If the systolic pressure goes too high or too low, a warning event (different than an ordinary systolic event) is fired to alert the nurses that a patient needs attention. We could define three different event classes: a `SystolicEvent`, a `DiastolicEvent`, and a `WarningEvent`. Very often, however, a handler interested in one of these events will be interested in multiple of them, so it make sense to define only one type of event, a `BloodPressureEvent`

4.5.1 New Event Classes and Interfaces

Central to this example is the need for a new type of event, a `BloodPressureEvent`. As we saw illustrated in Chapter 2, there are really a collection of classes and interfaces needed for each type of event. These include:

- The `BloodPressureListener` interface which defines the methods the event handlers must implement. These include methods for each sub-type of event. We will need methods declared to handle: changes to the systolic pressure, changes to the diastolic pressure, and warning events – where the systolic pressure goes out of range. We do not include the actual event handlers here, since application programmers will define the event handlers, specifying application-specific actions to take when a `BloodPressureEvent` fires.
- The `BloodPressureEvent` class which encapsulates all the critical data about the event. A `BloodPressureEvent` must contain, the event source, the changed value, and possibly flag telling whether the event is for the systolic or diastolic pressure, or a warning.
- The `Patient`, the event source, which fires the events.

4.5.2 The `BloodPressureListener` Interface

The `BloodPressureListener` interface specifies the methods that each handler must implement. We have three very closely related types of events that are being handled: changes to the systolic

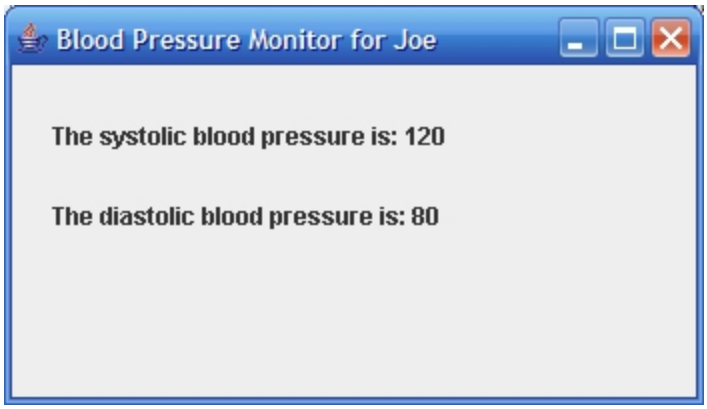

Figure 4.6: The blood pressure monitoring display at the nurse's station. When a warning occurs, a message box pops up, as well.

pressure, changes to the diastolic pressure, and warnings. Because there are three closely related types of events, the interface declares three methods: `systolicChange()`, `diastolicChange()`, and `warning()`. `BloodPressureListener` is given in its entirety below.

```
1 /**
2  * This interface defines the methods that must be provided by
3  * all objects receiving BloodPressure events.
4  *
5  * @author Stuart Hansen
6  */
7 public interface BloodPressureListener extends EventListener {
8     // receives the event if the upper value (systolic) changes
9     public void systolicChange(BloodPressureEvent e);
10
11    // receives the event if the lower value (diastolic) changes
12    public void diastolicChange(BloodPressureEvent e);
13
14    // receives the event if the systolic pressure goes extreme
15    public void warning(BloodPressureEvent e);
16 }
```

4.5.3 The `BloodPressureEvent` class

Each of the methods in the listener interface takes a `BloodPressureEvent` as a parameter. A `BloodPressureEvent` object contains the following information:

- the event source, in this case the patient.
- `AWTEvent`s include an integer event type. We will use this type field to keep track of whether we have a systolic, diastolic or warning event.

  These first two variables are inherited from `AWTEvent` and their values are set in the call to `super()`.
- the new value.

Generally speaking, `AWTEvent` objects are passive. They contain information used by handlers, but don't contain much active code. However, events are queued in the system event queue. To process the event, the system needs to go back to the source which contains the handlers. The easiest way to do this is to have the event class implement the `ActiveEvent` interface, which includes only one method, `dispatch()`. As discussed in the previous section, the `EventQueue`'s `dispatchEvent()` method checks to see if an event is an `ActiveEvent` event. If so, it calls the event's `dispatch()` method. In our `BloodPressureEvent` class, we will use the event's `dispatch()` method to call back to the `Patient`'s `handleBPEvent()` method.

```
 1 import java.awt.*;
 2
 3 /**
 4  * This class encapsulates all the information for
 5  * when the blood pressure changes
 6  *
 7  * @author Stuart Hansen
 8  */
 9
10 public class BloodPressureEvent extends AWTEvent implements ActiveEvent {
11     // We use these constants to sort out what type of Blood Pressure
12     // event occurred
13     public static final int SYSTOLIC_EVENT = 0;
14     public static final int DIASTOLIC_EVENT = 1;
15     public static final int WARNING_EVENT = 2;
16
17     private int value;  // the new value
18
19     // This is the only constructor.
20     public BloodPressureEvent (Object source, int type, int newValue) {
21         super(source, type);
22         this.value = newValue;
23     }
24
25     // returns the value
26     public int getValue()  {
27         return value;
28     }
29
30     // Handle the event by dispatching it to the source.
31     public void dispatch() {
32         BloodPressureEvent event = (BloodPressureEvent) this;
33         Patient pat = (Patient)getSource();
34         pat.handleBPEvent(event);
35     }
36 }
```

The `dispatch()` method is short enough to be very easy to read. It first finds the patient that is the source of this event. It then calls the patient's `handleBPEvent()` method.

The need for `dispatch()` and `handleBPEvent()` may seem confusing. Conceptually it isn't too hard to explain, however. Java's event infrastructure knows how to handle all the Java defined event types. `ActionEvent`s, mouse clicks and the like are understood by the `EventQueue`'s `dispatchEvent()` method. When `dispatchEvent()` comes to a `BloodPressureEvent` it doesn't

know what to do with it, however. Making our events `ActiveEvent`s solves the problem.

4.5.4 The `Patient` class

The `Patient` class is our event source. The patient is a fairly typical class that stores data about the patient. In addition, because it is an event source, it has methods to add and remove listeners, mechanisms to fire events, and methods to help process events.

We declare and instantiate a vector of listeners to hold our handlers:

```
29    private Vector<BloodPressureListener> bpListeners =
30                                          new Vector<BloodPressureListener>();
```

We declare two short methods to add and remove handlers from the vector:

```
60   // add a blood pressure listener to this patient
61   public void addBloodPressureListener (BloodPressureListener bpListener)  {
62       bpListeners.add(bpListener);
63   }
64
65   // remove a blood pressure listener from this patient
66   public void removeBloodPressureListener (BloodPressureListener bpListener)  {
67       bpListeners.remove(bpListener);
68   }
```

Events are fired when values are blood pressure values are changes. `setDiastolic()` calls `fireBPEvent()` which enqueues the event in the system event queue.

```
97     // set the current diastolic blood pressure
98     public void setDiastolic (int diastolic) {
99         int olddiastolic = this.diastolic;
100        this.diastolic = diastolic;
101        if (olddiastolic != diastolic) {
102            fireBPEvent(new BloodPressureEvent (
103                        this,
104                        BloodPressureEvent.DIASTOLIC_EVENT,
105                        diastolic));
106        }
107    }
108
109    // Firing the event requires enqueuing it
110    public void fireBPEvent (BloodPressureEvent e) {
111        Toolkit kit = java.awt.Toolkit.getDefaultToolkit();
112        EventQueue queue = kit.getSystemEventQueue();
113        queue.postEvent(e);
114    }
```

`BloodPressureEvent`'s `dispatch()` method calls the `Patient`'s `handleBPEvent()` method, so this also needs to be defined. Since this method handles all types of blood pressure events, we use a switch statement to distinguish them.

```
116     // Handle the blood pressure event when it comes out of the queue
117     public void handleBPEvent (BloodPressureEvent e) {
118         // clone the vector so registrations don't affect this event instance
119         Vector<BloodPressureListener> temp =
120                     (Vector<BloodPressureListener>) bpListeners.clone();
121
122         // Walk through the vector firing events to each listener
123         Iterator<BloodPressureListener> bplIterator = temp.iterator();
124         while (bplIterator.hasNext()) {
125             // The switch is needed to sort through the three types of
126             // blood pressure events
127             BloodPressureListener bpl = bplIterator.next();
128             switch (e.getID()) {
129                 case BloodPressureEvent.DIASTOLIC_EVENT:
130                     bpl.diastolicChange(e);
131                     break;
132                 case BloodPressureEvent.SYSTOLIC_EVENT:
133                     bpl.systolicChange(e);
134                     break;
135                 case BloodPressureEvent.WARNING_EVENT:
136                     bpl.warning(e);
137             }
138         }
139     }
```

4.6 The Beans-Like Implementation

The same blood pressure example can be written using beans-like events that avoid the AWT event queue. This is a worthwhile exercise, if for no other reason to gain insight into the the way the two systems work.

The main difference is that the event source now takes responsibility for processing events rather than relying on the system's event queue.

The `BloodPressureListener` interface remains unchanged from the previous example.

4.6.1 Revamped Blood Pressure Events

The `BloodPressureEvent` class is simpler than in the previous section.

- We no longer need the event type field to distinguish which type of event we are processing. Since the patient will be calling the handlers directly, and the patient already knows the event type, the information is redundant.

- The `dispatch()` method has disappeared. The `dispatch()` method provided a route back from the event queue to the source. Since we are no longer using the event queue, the method is not needed.

```
 1 import java.util.*;
 2
 3 /**
 4  * This class encapsulates all the information for
 5  * when the blood pressure changes
 6  *
 7  * @author Stuart Hansen
 8  */
 9
10 public class BloodPressureEvent extends EventObject  {
11     private int value;  // the new value
12
13     // This is the only constructor.
14     public BloodPressureEvent (Object source,  int newValue) {
15         super(source);
16         this.value = newValue;
17     }
18
19     // returns the value
20     public int getValue() {
21         return value;
22     }
23 }
```

4.6.2 The Revamped Patient Class

In this implementation, the `Patient` class takes full responsibility for invoking the event handlers. The `Patient` class includes methods to fire each of the different types of blood pressure events. The fire methods are called from within the `Patient`'s setter methods. Each setter checks to see if an event has occurred, and if so, it calls the appropriate `fire***Event()`. Below is `setDiastolic()`. `setSystolic()` is very similar, but may fire either warning or systolic change events.

```
89     // set the current diastolic blood pressure
90     public void setDiastolic (int diastolic) {
91         int olddiastolic = this.diastolic;
92         this.diastolic = diastolic;
93         if (olddiastolic != diastolic) {
94             fireDiastolicEvent(new BloodPressureEvent (
95                         this,
96                         diastolic));
97         }
98     }
```

Note that the code calls `fireDiastolicEvent()` only if there is a change in the value. This makes things slightly more efficient, since it avoids events that have no affect.

Below is the `fireDiastolicEvent()` method. It iterates across the list of listeners calling `diastolicChange()` on each.

```
132     // We call each of the handlers directly
133     public void fireDiastolicEvent (BloodPressureEvent e) {
134         // clone the vector so registrations don't affect this event instance
135         Vector<BloodPressureListener> temp =
136                         (Vector<BloodPressureListener>) bpListeners.clone();
137
138         // Walk through the vector firing events to each listener
139         Iterator<BloodPressureListener> bplIterator = temp.iterator();
140         while (bplIterator.hasNext()) {
141             BloodPressureListener bpl = bplIterator.next();
142             bpl.diastolicChange(e);
143         }
144     }
```

`fireSystolicEvent()` and `fireWarning()` are almost identical. They only vary in their names and in the listener method called.

4.6.3 Infrastructure or Client Code

The previous two sections have emphasized the differences in two Java based approaches to event processing. Both were really dealing with event infrastructure issues, however.

It is important to realize that client code is oblivious to these differences. The GUI and the event handlers are the same in either case. A programmer can use either set of Blood Pressure Event classes to implement a nurse's station.

4.7 Summary

As we saw in earlier chapters, it is possible to write event based applications with only a naive understanding of how events are processed. In this chapter we showed that an in-depth understanding of the event processing infrastructure is necessary. Making simplifying assumptions about event processing is fraught with danger, as minor differences in the processing can yield significant differences in the results.

We also showed how to use our understanding of the event processing infrastructure to develop our own event classes.

Chapter 5

Threads and Events

5.1 Introduction

This chapter introduces threads as they relate to events and event based programming. There are really three closely related topics covered. First, we will discuss how using threads can solve some of the problems we face with long running event handlers. Unfortunately, this solution introduces several new problems, and we will briefly discuss each of these. Finally, we will discuss threads as event based systems. That is, rather than using threads in our event based programs, we will consider a thread as an event based system.

5.2 Background

A *thread of execution* is the smallest unit of computation that can be scheduled to run by the operating system (OS) or virtual machine (VM). It is responsible for carrying out some part of a computation, possibly an entire program, or possibly only some small part of a program. Every program, even the simplest *Hello World!* program has a thread of execution that runs the program. More complex systems often have multiple threads running concurrently. The basic concept is illustrated in 5.1.

Threads are different than *processes*. A process may be thought of as an entire program in execution. By contrast, a thread is on a finer granularity. It may be a only very small portion of a program.

Programs in many modern languages divide memory up into three basic areas:

- The *code segment* contains the executable instructions for the program.
- The *heap* stores dynamically allocated objects. Any time the programmer says `new` in Java or `malloc` in C, memory is allocated from the heap.
- The *stack* stores local variables. The stack consists of *activation records*. Every time the program makes a method call, a new activation record is created and pushed onto the stack to store the methods variables. When a method returns, its activation record is popped off the stack.

A program's threads share the code segment and the heap. Each has its own stack. Thus, each thread has its own calling and return sequence, and its own local variables. For example, in Figure 5.1 Thread 1 has made calls to methods in Objects A, C, D and E, in that order. Its call stack will

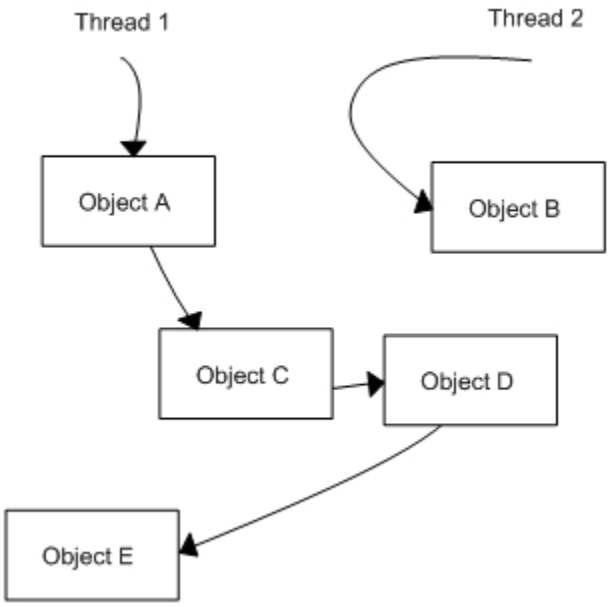

Figure 5.1: Multiple threads may be executing concurrently within the same application.

contain activation records for each of the calls. By contrast, Thread 2 has only made one call, to a method in Object B. Its call stack will only contain one activation record.

5.2.1 Threads and Concurrency

Conceptually, all running threads in a system are executing *concurrently*. We know that, in general, a computer doesn't have enough CPU power to execute all the threads at the same time, but we may think of them as running concurrently, while in reality, the operating system takes responsibility for swapping the threads in and out of the CPU, giving each a slice of processing time.

Many modern computers, including high end personal computers, contain multi-core processors. A core is the portion of the CPU that reads and executes program instructions. A single core processor reads and executes one instruction at a time. A multi-core processor can read and execute multiple instructions at a time, giving true concurrency. Each core reads and executes instructions independently of the others.

A multi-threaded program can run on a single core processor by swapping different threads into the processor, giving the illusion of true concurrency, see Figure 5.2.

A multi-threaded program running on a multi-core processor can achieve true parallelism, see Figure 5.3.

5.3 Why use Threads

Multi-threaded programs have several advantages over single threaded programs, including:

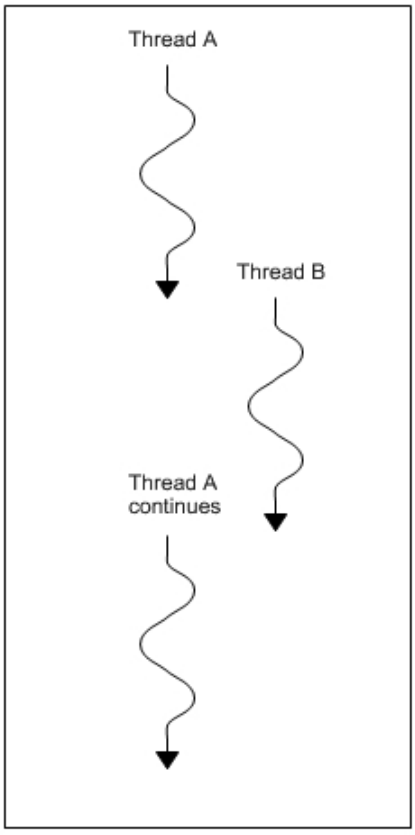

Figure 5.2: Threads alternate turns executing on a single core CPU.

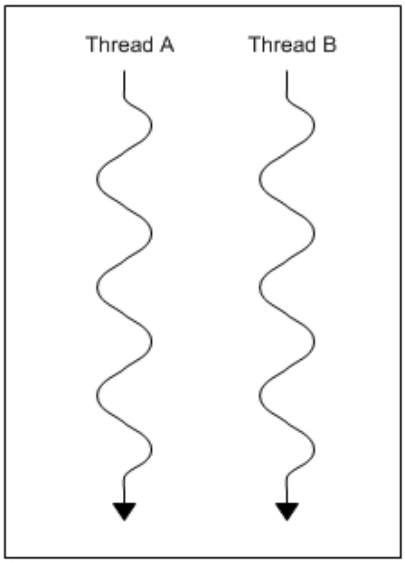

Figure 5.3: True concurrency can be achieved on multi-core CPUs.

Responsiveness

With a single thread of execution, the program behaves sequentially. It does one thing, then the next, then the next. If the user is waiting for the program, she must wait until the program gets around to responding. If the program is doing some heavy duty number crunching, accessing a remote database, there may be a significant delay. This is an important when dealing with GUIs or other interactive programs. If the delay is long enough, the user may think the program is hung and shut it down, while really it is just completing some processing.

Java, for example, uses a separate thread, named the *event dispatching thread* to handle GUI events. This lets the user continue to interact with the program, even when there is a delay in the event handling.

Java event dispatching thread only addresses part of the problem, however. Consider what happens if there are multiple handlers for the same event. Say, for example, you have a fire alarm system. A teacher, not wanting to go outside to smoke in the January cold, grabs a quick cigarette in the restroom and sets off the alarm[1]. Multiple things should happen simultaneously:

- The sprinkler system should spray water drenching the inveterate smoker and putting out his cigarette.
- The acoustic alarm should sound throughout the building, telling other teachers and students to exit the building a quickly as possible.
- The strobing visual alarm should be set off, giving hearing impaired students and teachers the same warning.
- The fire department should be notified.

[1]Possibly more realistically, a student wanting a few extra minutes to study for an exam intentionally pulls the fire alarm in the hallway.

We want all four responses to happen simultaneously. The system should not wait for the sprinkler system to complete before the acoustic alarm sounds. Nor should the acoustic alarm shut off before the strobing alarm begins. We definitely want the fire department to be notified as quickly as possible.

Threads are an ideal solution to making the system responsive, but, as we will see in the next section, in languages like Java we will need to manage them explicitly to take full advantage of their capabilities.

Efficiency

Ideally, there is the potential to speed up our computation by using multiple threads. If we can keep all the cores on the CPU actively executing our program, we can expect a speed up related to the number of cores. For example, if we have a quad-core processor, our program can run up to four times as fast.

There are a number of reasons why we never achieve this ideal case:

- First, when we write a program we may have no idea what kind of processor it will be run on. We might be developing it on our dual-core desktop, but the user may have a low end single core processor or a high end quad-core. This makes it impossible to guarantee to guarantee that we can take full advantage of all the cores.
- Second, even low end desktop computers run many programs simultaneously. Our program will be competing to use the CPU's cores, with no guarantee it can have all of them whenever it wants.
- Next, the OS or VM needs time to swap threads in and out of the CPU. If you have many active threads in your program, but are running on only a single core system, you may see a degradation in performance, rather than an improvement.
- Lastly, threads sometimes block. For example, if a thread is waiting for user input, it won't do any further computation until it gets the input. A thread may also be waiting for a resource being used by another thread. If we are not careful, this situation can lead to even more serious problems, like two threads waiting for each other, a problem known as *deadlock*.

Resource Sharing

An application's threads share the application's heap and all of them have access to the objects stored there. These objects may include data structures, open files and access to remote objects like databases. Having threads share these objects makes sense from the application's point of view, as something like a database of employee records may be needed for a number of different reasons by a number of different threads. Opening and closing databases can be time intensive, so sharing a database handle among the threads is an efficient use of resources.

5.4 A Multi-Threaded Event Handler

In the previous chapter we implemented a patient monitoring system to keep track of changes in a patient's blood pressure. Because we were implementing the event classes, not the client code, we didn't look at any of the event handlers. In this section, we return to that example, looking at how to implement the handlers. As you should recall, the handlers are executed by the event dispatching thread. In general, we can write handlers the same way we were doing it back in Chapter 2. That is, we implement the methods in the handler interface, taking care of the tasks required. However,

if there is the potential for a handler to block, or if a handler may run for a very long time, it is best to have the handler start a new thread to complete its tasks.

5.4.1 Simple Handler

The handler interface, `BloodPressureListener` contains three methods:
`public void systolicChange)(BloodPressureEvent e)`
`public void diastolicChange(BloodPressureEvent e)`, and
`public void warning(BloodPressureEvent e)`
one for each of the three possible causes of blood pressure events. We illustrate how to implement a handler just for the last one, *Warning* events.

```
66      // Processes warning events when systolic pressure is too high or too low.
67      public void warning (BloodPressureEvent e)
68      {
69           Patient patient = (Patient) e.getSource();
70           JOptionPane.showMessageDialog(
71                          null,
72                          "WARNING! Patient "  + patient.getName() +
73                          " has systolic pressure of: " + e.getValue());
74      }
```

Our handler is quite simple, it pops open a message box at the nurse's station telling them that the patient has a problem. There is an OK button on the dialog and the nurse clicks the button to close the dialog. The message box blocks the event dispatching thread, however, waiting for the nurse to click OK. This presents problems if there are other handlers for this event, e.g. notifying a doctor, or other, possibly more important events firing. They will be blocked until the nurse clicks OK.

5.4.2 Threaded Handler

The solution is to use another thread to process the warning event. The new thread should be started from within the handler. The event source should never be responsible for creating a new thread in which to execute the handler, since the source will not know if the handler even needs to be threaded, and unnecessarily creating a new thread each time a handler is called may itself result in system degradation.

The easiest way to have a handler's task execute in its own thread is to declare an inner class that implements the `Runnable` interface. There is one method in the `Runnable` interface, `run()`. Inside the `run()` method, place the handler's task code. The handler should then instantiate and start a new thread, using this inner class. The code snippet below shows how to do this for the `warning()` method. If this is the only handler for the event, you will not be able to discern and difference in how it runs for the previous section. If there are multiple handlers, however, this version will let those following this one begin running, without waiting for the message box to close.

```
66      // Processes warning events in a new thread
67      public void warning (BloodPressureEvent e) {
68          new Thread(new Warner(e)).start();
69      }
70
71      // We use a class implementing the Runnable interface
72      // to run the handler code in a new thread
73      private class Warner implements Runnable {
74          BloodPressureEvent e;
75          public Warner (BloodPressureEvent e) {
76              this.e = e;
77          }
78
79          // The thread's start method calls run
80          public void run() {
81              Patient patient = (Patient) e.getSource();
82              JOptionPane.showMessageDialog(
83                      null,
84                      "WARNING! Patient " + patient.getName() +
85                      " has systolic pressure of: " + e.getValue());
86          }
87      }
```

5.5 Problems Arising from Multi-Threading

It would not be appropriate for us to end this discussion on multi-threaded event based programming without some mention of the problems that may arise. Multi-threaded programming remains a bug prone domain. You are advised to be careful when implementing multiple threads to avoid as many of these problems as possible. While a complete discussion of these issues is beyond the scope of this book, a good operating systems text will explore these topics in more detail.

5.5.1 Problems with Resource Sharing

Threads share the code and heap of the application. If multiple threads wish to access the same resource, problems can arise. Consider, for example, the problem of two threads, [T1] and [T2], wishing to increment the same integer variable, `count`, simultaneously. That is, both threads wish to execute

```
count++;
```

If `count` starts at 0, the correct new value is 2. On the assembly language level, a typical sequence of instructions to accomplish increment is:

```
66   load count a1          ; load count into register a1
67   incr a1                ; increment a1
68   stor a1 count          ; store register a1 back into count
```

Let the two threads running on separate cores intermix the instructions as follows:

```
[T1] loads count into its local register a1              // value of count = 0
[T2] loads count into its local register a1              // value of count = 0
[T1] increments its a1 register giving a result of 1     // value of count = 0
[T2] increments its a1 register giving a result of 1     // value of count = 0
[T1] stores its a1 back into count.                      // value of count = 1
[T2] stores its a1 back into count.                      // value of count remains = 1
```

`count` was only incremented once!

The basic solution to this problem is to provide a locking mechanism, so that a thread can claim exclusive use of a resource until it has finished its task. Unfortunately, locks introduce still more threading problems.

Deadlock

Deadlock occurs if two (or more) threads each possess a lock on a resource, and are waiting for the lock on the other's resource. This cycle of waits cannot be broken by the threads, as they are not aware that it exists.

Starvation

If one thread holds a lock and another thread wants the resource, the second thread waits until the first thread has relinquished the lock before proceeding. If multiple threads are waiting for the same resource, only one thread gets the lock next and is allowed to proceed. Starvation may arise if threads are given priorities and the lock is next given to the thread with the highest priority. It is possible for a low priority thread to never gain the resource because there is always a higher priority thread in front of it. The thread never makes any progress.

5.6 Threads as an Event Based System

In the first part of this chapter we saw that threads are a useful programming tool to improve our event based programs' performance and responsiveness. In this section, we explore threads from a very different perspective.

Threads may be thought of as a type of event based system! This shouldn't surprise you too much. by now that we can think of the threading of a program as event based, while the remainder of the program may not be. This was the same situation we saw with GUIs. The I/O for the GUI was event based, while the rest of the program followed more traditional approaches. The biggest difference between the two is where the events originate. With GUIs, the events originated with the user. With threads, most events originate with the thread scheduler, the OS or VM, or the program itself.

There are numerous implementations of threads, and they vary in some important details. Java has its won threads. MS Windows implements threads. Posix includes PThreads, an effort to standardize threads across various Unix platforms. In this chapter, we will keep the discussion general enough to apply to most of thread varieties, most of the time.

What do we mean when we say that threads are an example of an event based system? We have seen how events are useful when developing GUIs. They let us implement application classes and event handlers separately from the GUI components, e.g. the event sources. The separation of responsibilities implied by M–V–C is a powerful approach to developing GUI systems.

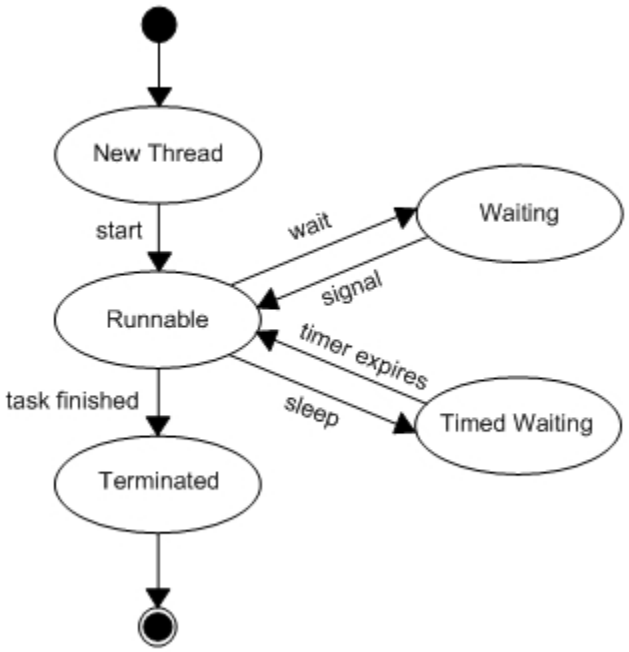

Figure 5.4: Threads have control state.

The same type of approach is useful whenever we need to decouple parts of a system. With threads, the decoupling is taking place between the application and the hardware on which it is running. As discussed earlier, threads need to be loaded into a CPU core to execute. At times, a thread also needs to be swapped out of a core, because it has exhausted its time allocation, or because it is waiting for a resource to become available. If other threads are waiting to execute, the current thread should give the CPU to one of the waiting threads.

Obviously, the OS or VM sits between the application and the hardware, and interacts with both. It takes responsibility for scheduling threads onto the CPU's cores. Hardware, OS and VM generate events that the threading system responds to.

In Chapter 1 we developed a conceptual model for event based systems. They had a number of different attributes. They were *state based*, contain *nondeterminism*, were *loosely coupled*, and had *decentralized control*. We discuss each as they relate to threads.

5.6.1 State Based

Figure 5.4 shows an overview of the various states of a thread.

A thread is created, in object-oriented programming typically through a call to a constructor. Creating the thread, however, does not automatically start is running. An additional call to a method named something like `start()` is generally required.

Once the thread is started, it may either be immediately assigned a processor core, or placed in a data structure, waiting for a core to become available. Similarly, while the thread is running, it may be asked to yield its core to another thread. This happens frequently in computing systems where there are many more processes and threads waiting to run than there are physical cores available. In our figure, a *Runnable* thread is any thread that is ready to run, whether currently running or waiting to run From the thread's point of view, it doesn't matter which, as it is ready to run in

either case.

While a thread is running, it may request a resource, e.g. open a file or connect to a database. This is done by calls to the operating system. These calls may take a significant amount of time for a couple of reasons. The resource may be slow by nature, e.g. reading from disk drives takes thousands of times longer than reading from memory; or the resource may be in use by a different thread, in which case our thread must wait for it to become available. In either case, the thread should enter a *Wait* state. Once the resource becomes available, the thread should be returned to the *Runnable* state. In Figure 5.4 the state in the upper right is for threads waiting for resources.

Sometimes it makes sense to wait for a specified amount of time. For example, in our stopwatch example in Chapter 3, we set timers to go off, updating the watches time. It may also make sense to have a combined state, where a thread returns to *Runnable* either when the requested resource becomes available or when a timer expires. A good example of this is a web browser. If a browser requests a page, it waits for the page to be loaded, or if no page is received after a given period of time it displays a default page with an error message. In Figure 5.4 *Timed Waiting* is the the lower right.

5.6.2 Nondeterminism

Nondeterminism means that it is impossible to determine exactly how a computation will proceed. The code a thread is executing tends to be deterministic. Our thread code is typically just ordinary Java, C++ or other code, the same type of programming we have seen since Computer Science 1.

Nondeterminism enters our programs only when there are multiple threads. Recall our incrementing `count` example from earlier in the chapter. We saw that there is the *potential* to get an incorrect result if two threads try to increment `count` concurrently. We might also get the correct result, however (see Exercise 3 at the end of this chapter).

The nondeterminism occurs because an executing thread can change state due to external events occurring at inopportune times, e.g. the OS or VM removes the thread from its core at a critical time during execution, giving the core to a different thread competing for the same resource.

5.6.3 Loose Coupling

Coupling refers to any way that objects interact within a computing system. Obviously, all parts of a system need to interact either directly or indirectly in order to accomplish a task, so coupling occurs. *Loose coupling* can occur in a couple of ways.

We can insert an additional object between two objects moving them further apart from each other. This is one of the roles that threads play when dealing with an application and an underlying OS or VM.

Decoupling Application from OS and VM

Threads provide a level of indirection between our application and the OS or VM. That is, we write our programs in a way that makes it looks like OS calls happen instantaneously, even though we know they don't. Reading from a file, or connecting to a remote database takes time, and typically our thread is removed from the CPU core to better utilize the core while our application waits. The application doesn't need to do anything to make the waiting work, however. The thread provides this service.

Thread Pooling

Another way that loose coupling can occur is to make the relationship between two objects dynamic, or changing over time. We have seen this repeatedly with the registration (and possible deregistration) of event handlers. Event handlers are registered with the event source at runtime. The action that a system takes when a button is clicked is determined by this registration. As we have seen, this is a looser from of coupling than direct method calls.

The same type of decoupling can happen with threads. The code a thread is to execute can be determined at runtime. We saw one way to do this in Java in Section 5.4, where a `Runnable` object was passed to the constructor of a `Thread`. This process can be more general than this, however. Threads can exist as objects without having application code assigned to them. When an application needs a thread, it can grab a free thread object, assign code to it, and start it running. This approach is known as *thread pooling*. Its primary advantage is that it saves on the overhead of creating (and later destroying) new thread objects every time one is needed by the application.

5.6.4 Decentralized Control

A thread is responsible for executing a section of code within our program. The thread has control over the execution, and in that sense, control is centralized.

This section isn't about who is controlling our program, however. It is about who is controlling our threads? Is there a centralized authority that decides when threads should change state? The answer is *No*. The thread is controlled from a variety of sources.

- The program is responsible creating and starting each new thread.
- The OS or VM maintains the list of *Runnable* threads and decides how to schedule onto CPU cores.
- A timer, under the control of the OS or VM, determines when a thread should relinquish its core.
- Other timers determine that a thread has waited long enough and return it to its *Runnable* state.
- OS or language libraries frequently determine that a thread should voluntarily relinquish its CPU core because it has made a request, say to open a file, that will take a long time to complete.
- Interrupt handlers tell the OS or VM that the request has been completed and that the thread should again be *Runnable*.

5.7 Summary

Threads play a central role in modern computing systems. Multi-core CPUs are becoming common place, making it possible for multi-threaded programs to be more responsive and efficient. Multi-threading poses challenges for programmers, as they need to understand locking mechanisms to prevent inappropriate resource sharing.

Threads may be thought of as a type of event based system. They are state based, contain nondeterminism, provide a way of decoupling an application from the underlying infrastructure, and are controlled in a decentralized fashion, all properties expected of event based systems.

Chapter 6

Distributed Event Programming

6.1 Introduction

Distributed computing involves multiple computers working together to accomplish some task. The computers communicate with each other by sending messages back and forth across a network. This model of computing is common in many real world applications. For example, browsing the Internet involves a computer running a web browser and web servers feeding it pages. In addition, web browsing requires other specialized services invisible to the user, but supporting the interactions, such as the domain name server (DNS) which translates the URL the user enters to a binary form, the IP address, that the network understands.

Another practical example of distributed systems are Enterprise Resource Planning (ERP) systems. Historically, different departments within a business purchased computing systems to help them with their particular part of the business, e.g. payroll systems, inventory systems, human resources systems, manufacturing systems, etc. These diverse systems need to work with each other many business related tasks. For example, the sales system needs to work with the inventory system to make certain there are enough of each product being purchases. ERP systems were developed to fill this niche. Now, it is rare to find a company that is not running an ERP system like SAP or JDEdwards.

6.2 The Big Picture

To understand distributed systems we start by looking at the big picture. There are three types of things that participate in a computer application: data, computation and users. All three can be distributed to various degrees.

Data and Information

Data may exist in a central repository, e.g. in a database, or on a file server. Remote computers can then access them, as needed, either by making a local copy or by directly accessing the storage on a remove machine. Data may also be distributed for a variety of reasons. For example, various groups may own various parts of the data, e.g. as in the example above, the payroll department or the inventory department own their own data, and store them on a machine local to their department.

Computer scientists have been distributing data for decades. Modern operating systems let us mount remote file systems and treat them as if they are local. Languages like Java let us open remote files and read from them using the same I/O classes as processing local files.

Information

The last decade has seen significant effort to structure data into *information*. Data are just values. Information, by contrast, places the values into a context. For example, 39 is a datum. <age>39</age> implies that we should treat 39 as an age. The *eXtensible Markup Language (XML)* and a variety of its derivatives, are currently the dominant way of structuring data into information. They use tags, as those shown here, to build hierarchical structures from the data. For example,

```
<person>
   <name>Erica Haller<\name>
   <age>39<\age>
   <profession>College Professor</profession>
<\person>
```

could be a simple XML structure describing a person.

If two separate computing systems are to access the information, they must first each understand the structure of the data. to meet this need, XML introduced *schema* which describe the structure of the data. Schema are XML documents that describe the structure of other XML documents.

Computation

Distributed computation takes place across spatially separated computing systems operating asynchronously. This is different than multi-threading discussed earlier in the text. Multi-threading gives us concurrency, but isn't considered distributed, as the threads share the application's code and heap. Distributed systems are more loosely coupled, communicating with each only via well defined protocols. For example, it is very common to have a database "back-end" running on a separate computer from the rest of the application. This allows extra computing power to be dedicated to running complex database queries and adds another level of security. The application sends a query to the database. The database executes the query and returns the result.

Distributed computation is still an important area in computer science research. It poses some ongoing interesting and challenging problems. For example, developing dynamic distributed systems, where services and devices come online and go offline as needed, is still very difficult. Some progress is being made. With protocols like Bluetooth, it is now possible for two devices to discover each other, once they are within reasonable proximity.

Users

It may seem odd at first to talk about users as being distributed, but in many cases the primary purpose of a distributed system is to coordinate distributed users. Online multiplayer games and business professionals holding online meetings are both examples. The most typical paradigm used for these applications is the client-server model, where each user interacts with client software, which, in turn, interacts with the server.

6.3 Distributed System Infrastructures

Obviously, if we are going to build distributed systems, we need hardware and software infrastructure to support the systems working together. A detailed understanding of networking hardware is beyond the scope of this text. For our purposes you may assume that the computers being used have a network card installed with a cable (or virtual cable) connected to it.

Since this is a programming text we do need to gain an understanding of a variety of types and levels of programming support supplied by various distributed computation infrastructures.

Low Level Primitives

At the lowest level, bytes of data are passed between two computers. Almost all programming languages include objects known as *sockets* to facilitate this computer to computer communication. A socket is simply a stream of data flowing between two computers. It is completely up to the developers to decide what data should be sent and received and how that data should be interpreted. On this level, the developers must deal with many subtle challenges. The program must potentially contend with different hardware and operating systems, different implementation languages, unreliable data communication, crash/recovery scenarios, security, and much more. For these reasons higher level abstractions are now commonly used.

Middleware

Middleware systems provide an infrastructure around which to build a distributed system. As the name *middleware* implies, the software sits 'between' the two communicating computers. Both systems must have the middleware installed. The application program then communicates with the middleware, which in turn takes over all low level details of communicating.

Middleware systems vary significantly based on the computing niche they are trying to serve. Here are a few examples:

The Common Object Request Broker Architecture (CORBA)
CORBA was designed to facilitate integrating *legacy systems*. Legacy systems are pre-existing systems that are still filling a useful role in an enterprise. Two legacy systems may have been written in different languages and run on different hardware with different operating systems. They may differ in how they represent numbers, e.g. big-endian vs. little-endian, or in the order parameters appear in a compiled method. The *Common Object Request Broker Architecture, CORBA*, abstracts away these nasty differences. A CORBA implementation supplies libraries that allow an application to communicate with an *Object Request Broker, an ORB*. Any system that can communicate with an ORB can communicate via the ORB with any other CORBA based system.

Web Services
The World Wide Web infrastructure is immense. There are millions of web servers running and millions upon millions of web browsers accessing them. Not only that – amazingly, it all works. A user sitting in Kenosha, WI can call up a web page on a server in Bangalore India, and it will arrive in just a few seconds. One of the things that makes the web work so well is that it is built on top of very well understood languages and protocols, e.g. *html* and *http*.

Web services take advantage of the web infrastructure to build distributed applications. Web services are programs that run on web servers. Because they are programs, they can provide more complex services than just returning simple web pages. Web services receive requests for information and return results in *XML* documents. Because HTML and XML are closely related tagged languages, the web infrastructure works well with both.

Jini
Jini is a Java based infrastructure developed to implement service oriented dynamic distributed systems. *Service* and *dynamic* are central to Jini's intended use. A Jini network is made up of services

that are used by clients. The service may be a chat service, or a toaster[1]. Dynamic means that the services availability can change. New services can join the network. Others may crash or have network failures. When a service comes online, it advertises itself. When a client wants to use an available service, it sends the service requests. Building robust, dynamic distributed systems is still one of the major ongoing challenges in distributed systems research. Jini is a step in that direction.

Advanced Infrastructures

There are dozens of other distributed system infrastructures. Some are production level and compete directly with those above. Other are research tools. Distributed systems programming is still a research field in academia, and more tools are always being developed.

6.4 Event Based and Distributed Systems

Conceptually, event based and distributed systems have much in common. They are both made up of different parts running fairly independently of each other. The parts communicate by sending discrete messages to each other.

More specifically, many of the properties discussed in Chapter 1 apply to both types of systems.

Loose Coupling

Agents in distributed systems are loosely coupled in ways very similar to the source handler coupling in event based systems.

Dynamic Binding

Event based systems register handlers at runtime. Distributed systems go through a setup phase similar to handler registration. In a distributed system, a program begins running on each system. Sometime thereafter communication links are established between them.

Nonblocking

In general, event sources do not block, but continue to run, not waiting for handlers to complete. Similarly, in many distributed systems messages may be sent to remote agents and the local process continues running, not waiting for the remote computer to complete or return a message. If a return message, e.g. a result, is expected, a common model is to listen for it using a separate thread.

Decentralized Control

We say that event based systems use decentralized control because an event can cause a handler to execute in object A and a different handler to execute in object B. Each of these can fire ten more events that run handlers in still different parts of the code. Each handler takes responsibility for carrying out some part of the overall computing task, but there is no centralized authority taking charge of completing the task.

Processing in distributed systems can be very similar. Messages are passed from system to system, each carrying out its responsibilities, but again with no particular system in charge. The message senders assume that the receiving agent knows what it is supposed to do.

[1] Toasters serve us toast, don't they?

Nondeterminism

Both types of systems contain nondeterminism. As we have seen, the order in which events are handled may not be the same as the order in which they were fired. It depends on the underlying infrastructure. The network introduces significant nondeterminism in a distributed system. There may be arbitrarily long delays between when the message is sent and when it is received. Similarly, the order the packets arrive may not be the same as the order in which they were sent. It depends on their route through the network and the network load.

6.5 Publish - Subscribe

There are several additional ways that distributed systems may be even more event driven. First, as introduced in Chapter 1, many distributed systems work on a publish-subscribe basis. That is, System A registers its interest in receiving information of Type X from System B. Any time this information becomes available, B sends it to A and all other registered subscribers. This interaction motif closely parallels what happens between sources and handlers in event based systems.

Remote Events

A few distributed system infrastructures, e.g. CORBA and Jini, support remote events. These services are the equivalent of the event infrastructure discussed in Chapter 3. Event sources, running on one system, fire events to handlers located on a different system. The event service serves as an intermediary. It receives the event from the source and queues it until the handler is ready to process it.

6.6 Challenges Developing Distributed Systems

Programming distributed systems poses numerous challenges. In many ways these challenges parallel those of developing event based systems.

Shared Resources

If you have multiple processes competing for resources, you introduce the possibility of *deadlock*, where each process is waiting for a resource another holds, so no one makes progress. These types of problems also exist in any multitasking operating system. Solutions can be implemented using semaphores and monitors, but designs need to be very carefully thought through.

Nondeterminism

In a distributed system there is no way of knowing how fast remote processes are executing or how long messages will take to be delivered. This means that messages may be sent and arrive in many different orders. Each process in the system should continue functioning correctly, regardless of the order the messages are received.

Unreliable Networks

Distributed systems depend directly on an underlying network to communicate. Networks are much more reliable than they were in the past, but failures still occur. *Packets*, messages between two nodes in our system, may be dropped. Robust distributed systems should be able to recover from dropped packets and temporary network outages.

Failure and Recovery

Computer hardware and software fails. Distributed systems should be able to recover from *transient failures*. By this, we mean that if a node fails and is brought back to life, the computation should be able to continue. Consider an RSS feed. Users subscribe to an RSS feed about some topic, say rugby news. Any time there is some new rugby news, the feed sends all registered users the story. If the RSS feed fails and recovers, users should not have to re-subscribe.

Security

Distributed systems contain more potential security risks than standalone applications. Each process in the application needs to be secured, as does the communication between them.

6.7 The CORBA Infrastructure

The authors chose CORBA as the distributed system infrastructure for their examples in this chapter. The *Object Management Group (OMG)* designed CORBA as a specification, not an implementation. Different vendors provide their own CORBA implementations. As such, CORBA is platform independent and avoids issues involving programming languages and operating systems. For example, an accounting system, written in COBOL, running on an IBM mainframe, can communicate using CORBA with a payroll system written in C++, running on a Linux machine.

CORBA has been around long enough to have many solid implementations. A CORBA implementation comes with Sun's Java SDK making it widely available. Our examples will use this Java implementation, but here are CORBA implementations for most operating systems and programming languages.

Like any major software technology, there is a lot to learn before you can claim to be proficient in it. CORBA is no exception, and the reader should not expect to be an expert CORBA programmer when they have finished reading this chapter. Introductory CORBA can be cookbooked, however. That is, the 20-30 lines of CORBA code that a process needs can be borrowed from other examples with only minimal changes. Also, CORBA doesn't require as much detailed technical knowledge to run as many of the others infrastructures. This makes it quicker for students to jump start their programming.

6.7.1 Services, Tools and Classes

Perhaps the easiest way to understand CORBA is to think about the CORBA infrastructure as consisting of three different types of entities: services, tools and classes. The *services* are predefined CORBA objects that we will run along with our application code. The *tools* are compilers and other assorted software that we will use while we are developing our programs. The *classes* are programming language classes that we will depend on within our code.

CORBA Services

The CORBA specification contains a dozen optional services that may be delivered by a CORBA provider. Here we will discuss only two:

- **Name Service**
  In any distributed system, agents need to find each other. A web browser needs to find the web server before it can download a page. An airline reservation system needs to find the database of flight information before it can make a reservation. CORBA is no exception. Our

code, a CORBA client, needs to find other CORBA objects before it can communicate with them. CORBA objects are located using their *Interoperable Object Reference (IOR)*. The IOR is a binary string containing, among other things, the IP and port where the service may be found. There are multiple ways a client may obtain a service's IOR:

- We can hardcode into our client the IORs for other CORBA objects. This has major drawbacks, however. If the service moves to another system, for example, you have to change your program, too..
- We can give the location of the remote CORBA objects as runtime parameters, or as program input. This, too, would work, but if our program needs to interact with many services, the input becomes excessive.
- The most elegant solution is to use the CORBA name service. The *name service* is a CORBA object that maintains a table of names and IORs. We can use CORBA's name service to keep track of where CORBA objects are located. CORBA objects register with the name service. Other CORBA objects query the name service which returns the IOR of the desired service.

The observant reader will notice immediately that the name service, by itself, does not completely solve the problem. We still need to find the location of the name service, and we can't use the name service to find it! The problem is much simpler, however, as we only need to find the name service, which can then provide us with the location of other the other CORBA objects. An easy solution to this problem is to pass the location of the name service as runtime parameters, which is the approach we take in our examples. More on what those parameters look like in the section below.

The name service delivered with Sun's CORBA implementation is the *Object Request Broker Daemon (orbd). orbd* is a standalone application that we will start at the command line, just as we will start our own processes.

- **Event Service**
  The CORBA *event service* provides a delivery mechanism for distributed events. Remote events behave much like local events. There are times when we don't need to call a method on a remote object, we just need to notify the object that something of interest has occurred. CORBA event sources fire their events to the event service, which takes responsibility to delivering them to the handlers. Unfortunately, Java's SDK does not provide a CORBA Event Service and it is not discussed further here.

CORBA Tools

In CORBA terms, the public interface accessible by other CORBA objects is known as as a *service*, and the objects that implement the service are known as servers. CORBA clients access the service by calling the server's methods[2].

CORBA defines a "meta-language", the *Interface Definition Language (IDL)* to represent the signatures of the methods a CORBA service provides. These are the methods that other CORBA objects can access remotely. IDL is necessary because the different CORBA objects may be written in different languages. IDL is the universal language for all CORBA implementations.

[2]Note, however, that there is nothing stopping a CORBA object from being both a client and server. That is, a CORBA object can both provide services to others and call on other servers. The only real requirement is that a CORBA service needs to have its public interface defined.

Compiling IDL is a two step process. First, IDL descriptions are compiled into source code of another high level language, in our case, Java. If two CORBA objects are written in different languages, however, then different IDL compilers are used to generate the appropriate source code in each language. The IDL generated source code is later compiled, along with the rest of the CORBA object, using a standard language compiler, e.g. g++ or javac.

The primary CORBA tool we will need to develop our applications is the *IDL compiler, idlj.* `idlj` comes with Sun's Java SDK.

CORBA Interfaces and Classes

The Java package hierarchy beginning `org.omg.*` contains literally hundreds of Java interfaces and classes for CORBA development. In one chapter, we can't hope to acquaint the reader with all of them, but it is possible to write CORBA based programs with a working knowledge of only a few.

- *Object Request Broker*
  At the heart of CORBA is the *Object Request Broker (ORB).* It is the ORBs that supply the communication infrastructure. The ORBs communicate with each other, and pass information on to our application objects. Every process in our application will be associated with an ORB. If our accounting system can communicate with an ORB and our human resources system can communicate with an ORB, then they can communicate with each other.

- *Portable Object Adapter*
  Portable Object Adapters (POAs) sit between the ORB and the service. They help the ORB call methods in the service and pass the results back to the ORB. Since each service will have its own collection of methods, POAs are specialized to a particular service.

Figure 6.1 shows the structure of a client/server call in CORBA. The developer writes the **Client** and the **Server**. The ORBs are supplied by CORBA. Stubs and POAs are generated by `idlj`. The stub implements the required methods, but does not carry out the operations itself. Instead, the stub uses the ORB to communicate with the server, where the work is done. The server provides concrete implementations of the methods. Results are returned along the reverse path to the client.

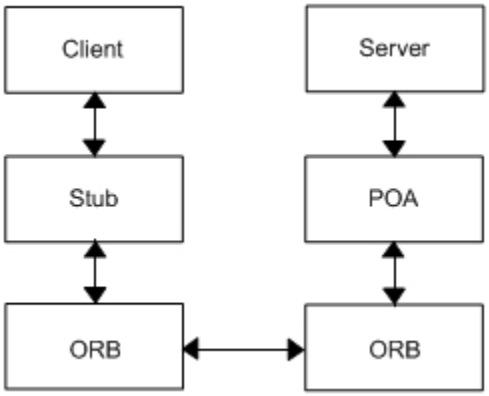

Figure 6.1: The main objects involved in invoking a remote CORBA service from a client.

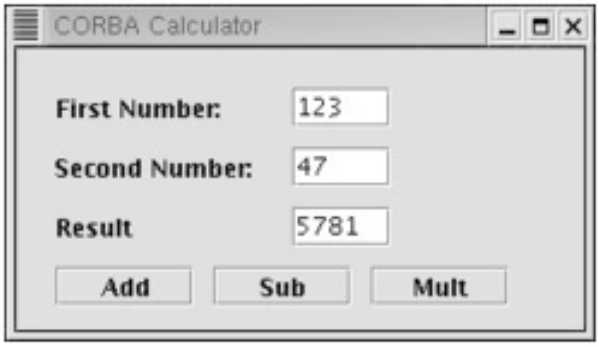

Figure 6.2: The Graphical User Interface for the calculator.

6.7.2 CORBA Programming Steps

The following steps are used to create and run CORBA based programs.

1. *Design the system*
   Designing a distributed system requires more work than a standalone application. The designer needs to decide different agents will be involved and what will be the responsibilities of each. The examples in this chapter are intentionally contrived to use a very small number of agents, but in the real world there may be hundreds of agents collaborating on some task, and the design decisions are far from trivial.

2. *Develop the IDL*
   Create and compile the idl description for each of the service agents. If you have done a good job in Step #1, this should follow easily.

3. *Implement the Agents*
   Implement the system agents, both the clients and the servers, using the interfaces and classes developed in Step #2.

4. *Start Everything Running*
   It is frequently the case in distributed systems that the various agents must be started in a particular order. With CORBA we will generally start the name service, then our servers, finally the clients.

6.8 Calculator Example

In the remainder of this chapter we present several small CORBA examples. Only illustrative code fragments are included. Complete source code for each of the examples is available from the text website.

Our first example is that of a calculator. In practice, there is really no point in having a distributed service to do simple math. However, a calculator is a well understood application so it serves as a nice introductory example.

Figure 6.2 shows the calculator's interface. The user enters two integers and the calculator performs one of three operations, addition, subtraction or multiplication, on them.

6.8.1 Defining the IDL

Our calculator service is simple. It provides three operations, addition, subtraction and multiplication. Below is our IDL file that contains the declares our service.

```
1 /* Calculator.idl
2  * This file contains the interface definitions for a simple calculator
3  *
4  * Written by: Stuart Hansen
5  *
6  */
7 module calculator {
8    interface Calculator {
9
10       // A method to add two numbers
11       long add (in long a, in long b);
12
13       // A method to subtract two numbers
14       long sub (in long a, in long b);
15
16       // A method to multiply two numbers
17       long mult (in long a, in long b);
18    };
19 };
```

IDL is a meta-language. It provides reserved words and data types that can be translated into high level programming languages. The IDL constructs in this example match fairly closely with Java's, but not exactly.

`module` translates to be `package` in Java.
`interface` remains `interface`.
`long` translates to Java's `int`[3].

The `in` in front of the `long` as part of a parameter declaration declares the parameter to be pass by value. That is, `a` and `b` are copied to the server, but are not returned. Java already uses pass by value for primitive parameters, so this presents no problems. Other legal parameter descriptors are `out` and `inout`. Java does not support copy out, or copy in and out parameters, so additional support classes called *holders*, are needed to use these. This example does not use `out` or `inout` parameters, so holders aren't needed.

6.8.2 Compile the IDL file

Java's IDL compiler, `idlj`, can be found in the `bin` directory of the JRE. It translates the IDL description into Java source code. It generates the stub and the POA classes discussed above as well as a few other interfaces and classes. The `idlj` command to translate this file is:

```
idlj -fall Calculator.idl
```

The `-fall` option tells `idlj` to generate `all` the CORBA classes needed to complete both the client and the server. The files generated are:

- `CalculatorOperations.java` and `Calculator.java` are interfaces containing the Java declarations for the Calculator. The contents of `CalculatorOperations.java` are:

[3] If you want a Java `long`, use IDL's `long long`.

```
1 package calculator;
2
3
4 /**
5 * calculator/CalculatorOperations.java .
6 * Generated by the IDL-to-Java compiler (portable), version "3.1"
7 * from Calculator.idl
8 * Monday, February 2, 2009 9:18:49 AM CDT
9 */
10
11 public interface CalculatorOperations
12 {
13
14   // A method to add two numbers
15   int add (int a, int b);
16
17   // A method to subtract two numbers
18   int sub (int a, int b);
19
20   // A method to multiply two numbers
21   int mult (int a, int b);
22 } // interface CalculatorOperations
```

Note how neatly each of the idl methods translated to a Java method.

`Calculator.java` contains an interface that extends `CalculatorOperations` and CORBA's `Object` class. Our calculator client will access methods from an object that implements `Calculator`, since it will implement `add()`, etc. and will be a CORBA `Object`.

- `CalculatorPOA.java` contains an abstract class that implements `CalculatorOperations`. It does not implement any of the methods from `CalculatorOperations`, however. To complete our server, we will extend `CalculatorPOA` and define the methods.

- `CalculatorHelper.java` contains some useful static methods.

- `_CalculatorStub.java` contains some CORBA methods to facilitate communications on the client end. Client code will not interact with it directly however, but will do so through the `CalculatorHelper` class.

- `CalculatorHolder.java` is a wrapper class used for `out` and `inout` parameters. Since our example contains neither, this class is not used.

6.8.3 Completing the Server

To complete coding the server, we will extend `CalculatorPOA`, implement the methods required by `CalculatorOperations` and write a constructor that does some CORBA setup.

```
 1 import calculator.*;
 2 import org.omg.CORBA.*;
 3 import org.omg.CosNaming.*;
 4 import org.omg.PortableServer.*;
 5 import org.omg.CosNaming.NamingContextPackage.*;
 6
 7 // This class implements a very simple CORBA server.
 8 // It provides add, sub, and mult methods that
 9 // clients may invoke.
10 //
11 // This Server requires that a CORBA Name Service be running.
12 //
13 // Written by: Stuart Hansen
14 // Date: February 2, 2009
15 public class CalculatorServer extends CalculatorPOA {
16
17     private ORB orb;  // the orb for this server
18     private POA rootpoa; // the root POA for this server
19
20     // The constructor sets up the ORB for communication
21     public CalculatorServer (String [] args)
22     {
23         setUpServerORB(args);
24     }
25
26     // implement the add method
27     public int add(int a, int b)
28     {
29         return a + b;
30     }
31
32     // implement the sub method
33     public int sub(int a, int b)
34     {
35         return a - b;
36     }
37
38      // implement the mult method
39     public int mult(int a, int b)
40     {
41         return a * b;
42     }
```

Lines	Commentary
1–5	Import various CORBA packages which contain supporting classes.
17–18	Declare the `ORB` and `POA` variables that we will use to facilitate communication.
20–24	Define the constructor which calls a setup routine to initialize our ORB.
26–42	Define the service methods declared in the idl and `CalculatorOperations`.

Setting up the CORBA Communication

The final programming step to complete the server is to set up the CORBA communications. Below is the listing of the method, `setUpServerORB()`, that accomplishes this. `setUpServerORB()` may at first seem complex with obscure CORBA method calls, but we will analyze it line by line, to show how it accomplishes its tasks. For those readers not interested in the details, the same method can be used for almost any server side application, just replacing all references to "Calculator" with the name of your server.

```
44
45      // This method setups the Server ORB
46      public void setUpServerORB(String [] args)
47      {
48          try{
49              // create and initialize the ORB
50              orb = ORB.init(args, null);
51
52              // get reference to rootpoa & activate the POAManager
53              rootpoa = POAHelper.narrow(
54                  orb.resolve_initial_references("RootPOA"));
55              rootpoa.the_POAManager().activate();
56
57              // Convert our server to a CORBA object and IOR
58              org.omg.CORBA.Object ref =
59                  rootpoa.servant_to_reference(this);
60              Calculator calc = CalculatorHelper.narrow(ref);
61
62              // Look up the Name Service
63              org.omg.CORBA.Object nameServiceObj =
64                orb.resolve_initial_references("NameService");
65              NamingContextExt nameService =
66                  NamingContextExtHelper.narrow(nameServiceObj);
67
68              // Bind (register) the Calculator Server with the Name Service
69              String name = "CalculatorServer";
70              NameComponent path[] = nameService.to_name( name );
71              nameService.rebind(path, calc);
72
73              System.out.println("Calculator Server Ready");
74
75              // wait for client requests
76              orb.run();
77
78          } catch (Exception e) {
79              System.err.println("ERROR: " + e);
80              e.printStackTrace(System.out);
81          }
82      }
83
84      // The main program simply creates a new Calculator Server
85      public static void main(String [] args) {
86        new CalculatorServer(args);
87      }
88  }
```

Lines	Commentary
45	The parameters coming into `setUpServerORB()` come from the `main()` method and contain the location where the name service, `orbd`, is running. See the section below on starting the application.
49	Initialize the ORB. We pass along the name service location to the ORB. In our case, the second parameter is `null`, but could contain a list of application specific properties. Note that you do not construct an ORB, only call its `init` method.
52-54	Set up the POAs for our system. POAs exist in a hierarchical structure. For this example we really don't need to worry about multiple POAs, but there is one `rootPOA` for our server.
52-53	Ask the ORB to give us a reference to the `rootPOA`. Wrap this request with a call to `narrow()`. Narrowing is CORBA's equivalent of casting to a more specific type. Line 53 returns a CORBA object. `narrow()` converts it to a `rootPOA`.
54	Activate the rootPOA's manager.
56-70	Register our server with the name service.
56-59	Convert our server (`this`) to a CORBA object, using the `rootPOA`.
62-65	Request a reference to the name service from the orb and again narrow it to be a name service (`NamingContextExt`) object.
68-70	Register our server with the name service.
75	Place the ORB into listening mode, actively waiting for requests from clients. Note that the ORB executes `run()` in the current thread. Because `run()` waits for clients to send requests to the server, any code following `run()` will not be reached.
77–80	Handle exceptions. Most exceptions that occur when starting out programming CORBA will be for null references. For instance, if the name service is not running, `orb.resolve_initial_references("NameService");` will return `null`. An exception will be thrown on the next line, where we try to narrow it to be a `NamingContextExt`. The problem is not with the narrowing, but with the null reference.
83–86	The `main()` method simply creates a `CalculatorServer`.

6.8.4 The Client

Much of the code for the client is GUI programming, which while event based, is not of interest to us in this chapter. We show only the portions of the code that are central to our understanding of CORBA.

```
 1 import calculator.*;
 2 import org.omg.CORBA.*;
 3 import org.omg.CosNaming.*;
 4 import org.omg.CosNaming.NamingContextPackage.*;
 5
 6 import java.awt.*;
 7 import java.awt.event.*;
 8 import java.util.*;
 9 import javax.swing.*;
10
11 // This class implements a simple calculator client program.
12 // It relies on a CORBA server to do the actual calculations
13 //
14 // Written by: Stuart Hansen
15 //
16 public class CalculatorClient extends JFrame {
17
18     private Calculator calc;  // the remote calculator
19
20     // The GUI Components
21     private JLabel firstLabel, secondLabel, resultLabel;
22     private JTextField first, second, result;
23     private JButton addButton, subButton, multButton;
24
25     // The constructor sets up the ORB and the GUI
26     public CalculatorClient(String [] args)
27     {
28         setUpORB(args);
29         setUpGUI();
30     }
```

Lines	Commentary
18	Declare a `Calculator` object. It looks like a local object and its methods will be invoked just like a local object. The only difference will be when we instantiate it.
28	`setUpORB()` is shown below.
29	`setUpGUI()` is omitted to save space. The complete listing may be found the text's website.

```
88     // An inner class to handle clicks on the add button
89     private class AddHandler implements ActionListener
90     {
91         public void actionPerformed (ActionEvent e)
92         {
93             int firstNum = Integer.parseInt(first.getText());
94             int secondNum = Integer.parseInt(second.getText());
95             int resultNum = calc.add(firstNum, secondNum);
96             result.setText(Integer.toString(resultNum));
97         }
98     }
```

The interesting part of this handler is line 95 where we call `calc.add()`. As we will see below `calc` is a reference to a CORBA object that handles all the communication with the server, but it looks completely like a local object. All the communication with the server is completely hidden. Subtraction and multiplication handlers follow the same pattern and are omitted.

```
124     // Set up the data communication with the server
125     private void setUpORB(String [] args)
126     {
127         try {
128             // Create and initialize the ORB
129             ORB orb = ORB.init(args, null);
130
131             // Get a reference to the name service
132             org.omg.CORBA.Object nameServiceObj =
133                 orb.resolve_initial_references("NameService");
134
135             // Narrow (cast) the object reference to a name service reference
136             NamingContextExt nameService =
137                 NamingContextExtHelper.narrow(nameServiceObj);
138
139             // Get a reference to the calculator from the name service
140             org.omg.CORBA.Object calcObj =
141                     nameService.resolve_str("CalculatorServer");
142
143             // Narrow (cast) the object reference to a calculator reference
144             calc = CalculatorHelper.narrow(calcObj);
145
146         } catch (Exception e) {
147             System.out.println("ERROR : " + e) ;
148             e.printStackTrace(System.out);
149         }
150     }
151
152     // The main simply makes a new calculator client
153     public static void main(String args[]) {
154         new CalculatorClient (args);
155     }
156 }
```

Setting up the client's ORB follows a slightly different pattern than setting up the server's. The client does not register with the name service, since it is not receiving any calls from other objects. For the same reason, its ORB will not be waiting to receive requests. On the other hand, the client has to use the name service to find the calculator server.

Lines	Commentary
129	Initialize the ORB.
131–137	Use the ORB to find the name service and narrow it to be a `NamingContextExt`, a name service object. This code parallels that found in the server.
139-143	Use the name service to find the calculator server and narrow it to be a `Calculator`.

6.8.5 Starting the Application

Three separate processes must be started to run our application, the name service, the server and the client. They must be started in the order shown below.

```
orbd -ORBInitialPort 1060
java CalculatorServer -ORBInitialPort 1060
java CalculatorClient -ORBInitialPort 1060
```

`orbd` is the Java's CORBA name service. The `-ORBInitialPort 1060` specifies which IP port that should be used when the client and server communicate with the name service. Any unused port number will work. Some operating systems block lower numbered ports, so choosing a number above 1024 is appropriate.

In the lines shown above, all three programs are running on the same machine. The name service, server and client may also run on different machines. In this case use `-ORBInitialHost` when starting the server and client, and specify the remote machine where the name service is running. For example, if `orbd` is started on a machine named `onion.cs.uwp.edu` and the server and client are started on `bratwurst.cs.uwp.edu`, then the commands will be:

On onion:

```
orbd -ORBInitialPort 1060
```

On bratwurst:

```
java CalculatorServer -ORBInitialHost onion.cs.uwp.edu -ORBInitialPort 1060
java CalculatorClient -ORBInitialHost onion.cs.uwp.edu -ORBInitialPort 1060
```

6.9 Asynchronous Method Invocation

The previous example illustrated a client-server application. The client blocked on each call to the server, waiting for results. This made sense, as we didn't want to move on to a new calculation until we obtained results from the previous on. The same blocking model is used for most client-server interactions. A web browser waits for a web server to return a page. An ftp client waits for an ftp server to download a file it requests.

There are nonblocking models for distributed computation, as well. Messages are sent between agents, but the sender does not wait for a response. The nonblocking models are more event like. In event based programming our event sources do not block waiting for the handlers to complete. In nonblocking distributed systems, the message sender does not block waiting for a response from the message recipient.

6.9.1 CORBA and Asynchronous Messages

CORBA's IDL declares asynchronous methods as `oneway`. A `oneway` method has no return value and has no `out` or `inout` parameters. The call is made to the server and the client continues running, not waiting for results. When a `oneway` method is invoked, the server acts much as an event handler does in a standalone event based application. It receives the message. It runs the method specified (equivalent to an event handler) and does not return any value to the client. Amazingly, the only change needed in our coding style is to include the word `oneway`.

6.9.2 Heart Monitor Example

Heart patients in hospitals are frequently hooked up to monitors to keep track of heart rate, blood pressure, or any of a number of other critical values. The monitors relay signals down to the nursing station, so a small team of nurses can easily keep track of all the patients on a wing. The software running at the nurses' station is the server, receiving messages from the patients, a.k.a. the clients.

Below is a simple idl file for such a system.

```
 1 /* Heart.idl
 2  * This file contains an idl description of a heart rate monitor
 3  * at a nurses station.  It receives messages from local monitors in
 4  * each patient's room.
 5  *
 6  * Written by: Stuart Hansen
 7  *
 8  */
 9 module heartMonitor {
10    interface HeartRateMonitor {
11
12       // This method registers a local monitor with the server
13       oneway void register (in string name);
14
15       // This method unregisters a local monitor with the server
16       oneway void unregister (in string name);
17
18       // A method to receive a new heart rate value
19       oneway void newRate (in string name, in long rate);
20
21       // A method to sound an alarm when the heart rate goes too fast or
22       // too slow.  It is left to the local client to decide what the
23       // appropriate values for the alarm are.  Typical values in a real
24       // hospital setting would be 40 and 140.
25       oneway void soundAlarm (in string name, in long rate);
26    };
27 };
```

Lines	Commentary
13	`register()` is called when a patient is initially hooked into the system. It informs the nursing station monitor that data will be arriving for this patient.
16	`unregister()` is called when the patient is taken off the system.
19	`newRate()` sends the current heart rate to the nursing station monitor.
25	`soundAlarm()` sends a message that the patient's heart rate is too high or too low.

The only new idl type in this example is string. An IDL `string` translates directly to Java's `String`.

The text's website contains complete code listings for `HeartPatient.java` and `NursingStation.java`. The classes are left intentionally simple, in order to emphasize their CORBA aspects.

6.10 Peer to Peer Distributed Applications

Our heart monitor example introduced asynchronous messages, but was still a client-server application. The patients (the clients) sent messages to the nurses' station, the server. In peer to peer applications all objects can both originate and receive messages. One way to think of peer to peer applications is that the objects will behave as both clients and servers.

6.10.1 CORBA Peer to Peer Applications

If all objects behave completely identically a single idl file describing their methods will suffice.

Frequently, however, even though all objects can both originate and receive messages, the communication remains asymmetric, with different objects sending and receiving different messages. This type of model requires multiple CORBA interfaces, one for each role in the system.

6.10.2 Chat Example

Consider a typical chat system. Users log onto a chat server. While logged on, they can send messages to the other users via the server. They can also receive messages from the other users, again via the server. Two interfaces are needed, one for the clients and one for the server, because both types of objects are sending and receiving messages.

chat.idl

```
1  /* chat.idl
2     Written by: Stuart Hansen
3
4     This file contains the idl code for a CORBA chat application.
5  */
6  module chatpkg {
7
8     // The Message struct contains all information for passing
9     // messages to the server and from the server to the clients.
10    struct Message {
11       string sender;
12       string messageText;
13    };
```

```
14
15    // A ChatClient sends and receives messages only with the server.
16    // Other client communicate with this client via the server.
17    interface ChatClient {
18
19       // Receive a message that a client is registered with the server
20       oneway void newClientNotification (in string clientName);
21
22       // Receive a message that a client has deregistered with the server
23       oneway void clientUnregisteredNotification (in string clientName);
24
25       // Receive a message from another client via the server
26       oneway void sendMessageToClient (in Message message);
27
28       // Receive a whispered message from another client via the server
29       oneway void whisperMessageToClient (in Message message);
30    };
31
32    // The ChatServer has methods to register/unregister ChatClients
33    // and to receive messages from Clients.
34    interface ChatServer {
35
36       // Register a new ChatClient
37       //   return values
38       //     0 = taken
39       //     1 = ok */
40       long register (in string clientName);
41
42       // Unregister a ChatClient
43       oneway void unregister (in string clientName);
44
45       // Post a message to all registered clients
46       oneway void post (in Message message);
47
48       // Whisper a message to one other client
49       oneway void whisper (in string clientName, in Message message);
50    };
51 };
```

Lines	Commentary
10–13	Define a `struct`, which is a new idl entity in this example. A struct is a class that contains only public data. There are no methods and no private data. Structs translate to Java classes where all data elements are public. In this case we use it to define a data structure to hold messages. Messages will be objects of type `Message`. They contain both the name of the sender and the text. The Message class is created when we compile our idl file using `idlj`:
17–30	Define the chat client's interface. Note that some of the method names may be a bit confusing. Recall that the `ChatClient` methods will be implemented by the clients, but called by the server. We chose to name them from the server's perspective. For example, `oneway void sendMessageToClient (in Message message);` is a method within the client class that the server will call when it wants to send a message to the client. From the client's point a name like `receiveMessage()` may be a better name.
34–50	Define the chat server's interface. A chat client sends a message to all other clients by sending it to the server's `post()` method for distribution.

```
 1 package chatpkg;
 2
 3 /**
 4 * chatpkg/Message.java .
 5 * Generated by the IDL-to-Java compiler (portable), version "3.2"
 6 * from chat.idl
 7 * Tuesday, February 3, 2009 1:24:27 PM CST
 8 */
 9
10 public final class Message implements org.omg.CORBA.portable.IDLEntity
11 {
12   public String sender = null;
13   public String messageText = null;
14
15   public Message ()
16   {
17   } // ctor
18
19   public Message (String _sender, String _messageText)
20   {
21     sender = _sender;
22     messageText = _messageText;
23   } // ctor
24 } // class Message
```

Client Connection to the ORB

The code in the chat clients to connect to their ORB and the server is a a combination of what was previously in the client and server classes. Code from a typical client is shown below.

```
231         // We run the orb waiting method in a separate thread so that we can
232         // continue processing in the main thread
233         public void run ()
234         {
235             orb.run();
236         }
237
238         // use NameService to connect to time server
239         private void connectToChatServer( String [] params )
240         {
241             try {
242                 // Initialize the orb
243                 orb = ORB.init( params, null );
244
245                 // Connect to the name service
246                 org.omg.CORBA.Object corbaObject =
247                         orb.resolve_initial_references("NameService");
248                 nameService = NamingContextExtHelper.narrow( corbaObject );
249
250                 // Look up the chat server
251                 NameComponent nameComponent =
252                                 new NameComponent("ChatServer", "" );
253                 NameComponent path[] = { nameComponent };
254                 corbaObject = nameService.resolve( path );
255                 chatServer = ChatServerHelper.narrow( corbaObject );
256
257                 // Register this object with both the name service and
258                 // the chat server
259                 POA rootpoa = POAHelper.narrow(
260                         orb.resolve_initial_references("RootPOA"));
261                 rootpoa.the_POAManager().activate();
262
263                 org.omg.CORBA.Object ref =
264                      rootpoa.servant_to_reference(this);
265                 ChatClient meIOR = ChatClientHelper.narrow(ref);
266
267                 NameComponent path2[] = nameService.to_name( name );
268                 nameService.rebind(path2, meIOR);
269
270                 chatServer.register( name );
271
272                 // Use a separate thread to start the orb
273                 new Thread(this);
274             } catch (Exception e) {
275                 System.out.println ("Unable to connect to chat server");
276                 System.out.println (e);
277             }
278         }
279     }
```

Clients need to register with both the name service and the chat server.

Lines	Commentary
233–236	Used to create a new thread for the ORB, so that the main thread does not block. Called when the new thread is started on line 273.
245–255	Resolve the name service and chat server to CORBA objects.
257–270	Carry out the registrations.
273	Start the ORB running in a new thread.
233–236	Clients also need to call `orb.run()`, as the server did, so they can receive messages. `orb.run()` executes in the current thread and blocks, however. Because clients need this thread to continue their own processing, so for example, they can originate messages, a separate thread is started.

The complete listing for the chat server is found on the textbook's web site.

6.11 Conclusion

This chapter introduced you to distributed programming and some of the basic paradigms, e.g. client–server and peer–to–peer, that are frequently used when building distributed applications. The distributed infrastructure we used to develop our examples was CORBA. There are certainly many other infrastructures available. In the next several chapters we will continue our discussion of distributed event based programming, but using the World Wide Web as the infrastructure.

Chapter 7

Events and the Web

7.1 Introduction

In this chapter we discuss event based programming and the World Wide Web. In recent years the web has become ubiquitous. There are millions of millions web pages available to computer users around the world. Even new verbs, e.g. "to google", have entered the lexicon, as a result of the web. But how does it relate to event based programming? There are several ways:

- **GUI Structures on Web pages**
  Scripting languages like `Javascript` make is easy to include GUI components such as buttons and textboxes on web pages. In recent years high end IDEs like NetBeans and Visual Studio make it possible to create these web pages in a drag and drop fashion, without even needing to know scripting.

- **Web Applications**
  A *web application* is a computer program whose primary interface is made up of web pages. The program is running on a server, and the user interacts with it through one or more web pages. You don't have to look far to find web applications. Most major email systems come with a web client. Similarly, as college professors, the authors regularly look up student records on-line. If you visit `http://yahoo.games.com`, you will find dozens of games with web interfaces. Universally, all of these systems are event based in their interactions with the server. They follow the *Request-Response* interaction model discussed in Chapter 1.

- **Web Services**
  Web applications are great. We can take our model a step further, however. Why limit ourselves to just interacting with them via web pages? A *Web Service* is a service oriented program running on a web server, but the client may be a web browser, a standalone program, or another web server. Web services let companies produce and sell services via the web. The classic example of a web service is *PayPal*. *PayPal* securely manages the financial portion of sales, so users no longer have to enter their credit card numbers on-line to make purchases. Both the consumer and the seller interact with PayPal to complete the transaction.

The World Wide Web is obviously a loosely coupled distributed system and both web applications and web services are distributed in nature. To a certain extent, then, this chapter is just another chapter about distributed programming, just using a different distributed infrastructure. There are good reasons to use the web infrastructure to develop distributed systems. The primary reason is that web is fairly mature and the protocols and languages used to communicate between browsers

and servers are very well understood. This makes it easier for developers to create and debug their applications.

7.1.1 Historical Perspective

Although the basic concepts of hypertext and networking extend back farther, the World Wide Web originated in 1990 when two scientists working at CERN in Switzerland, Sir Tim Berners-Lee and Robert Cailliau, proposed building a network storing hypertext pages that could be viewed by browsers. The Web really took off when the Mosaic Web browser was developed in 1993 by Marc Andreesen and others at the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign.

Early web pages were *static*. A web designer created an `html` document and deployed it on a web server. Users browsed to the page and read the document's contents. The page didn't change unless the web designer deployed an updated version. It soon became clear that the static model was inadequate because content on the web changed. By 1995 Netscape had introduced *server side pushing* of new content to browsers[1]. This allowed the server to update browser pages at its discretion. JavaScript was introduced in 1996. JavaScript is embedded in web pages and allows web pages to modify themselves by executing code as needed.

Today, the Web is very dynamic. Web servers respond to users not only with static html pages, but also with content created on the fly, primarily in response to information sent with requests from the user. *Common Gateway Interface (CGI)* scripts, *Servlets*, *Java Server Pages (JSPs)*, *PHP*, *Ruby on Rails*, and a variety of other technologies have emerged that let programs execute on web servers and produce dynamic content.

7.1.2 Multi-tiered Architectures

Many real world web applications use a multi-tiered architecture, where web pages serve as the user interface, the web server contains application code, a.k.a. business logic, and there is a database back-end, which stores persistent information for later reference.

In this model the browser serves as a *thin client*, with limited responsibilities. It provides a user interface and validates data before sending it to the server. Local data validation is important, because the web server may be remote and there can be noticeable time lag when interacting with it. Local validation saves time and server-side CPU cycles.

The server executes the application code. It runs programs that process orders, update inventory, or generate student record reports.

The database back-end serves as persistent storage. Orders for good, inventory, or student records all have very long lifetimes. The database stores this information for future reference.

This chapter does not contain multi-tiered architecture examples. While it is an interesting topic, they are not at the heart of understanding events and the web.

7.2 Web Fundamentals

Before we can intelligently develop web applications, we need to understand the fundamentals of how the web works.

[1] Note the use of event based terminology.

7.2.1 HyperText Transfer Protocol (HTTP)

For two computing systems to communicate they need an agreed upon set of standards for their messages. For example, in the previous chapter we briefly discussed IIOP, the Internet Inter-Orb Protocol, which CORBA uses. In fact, the Internet is based on a layered set of protocols, from the physical layer, made up of integrated circuits and wires, at the low-end, to the application layer at the high-end; all of which must work together to facilitate intercomputer communication.

The web uses *HyperText Transfer Protocol (HTTP)*. It is an application level protocol. It was designed to facilitate communication of *hypertext* documents [2]. Its most common use is to transfer web pages between a server and web browser. HTTP is based on request-response interactions, where a web browser requests a page and a web server responds with the page. For example, HTTP defines the standard for *Uniform Resource Locators (URLs)* which are entered after the `http:` in a web browser.

HTTP also defines the services that may be requested from the server. There are eight. Three of are interest to us: *Head* and *Get*.

- **Get**
  `Get` requests a web page from the server. Ideally, `Get` should not change the state of the web server. That is, multiple calls to `get` the same web page should always return an identical page.

- **Head**
  `Head` functions much like `get`, but does not request the entire page. A web page has a number of parts, including a heading and a body. The heading contains summary information about the page, but not the page content. `Head` only requests the heading information for the page. `Head` is typically used to test a link before an entire document is requested.

- **Post**
  `Post` submits data to the web server. Functionally, *Post* is similar to *Get*, in that a request is sent from a browser and a response is returned by the server. Because *Post* is meant for data submission, it is acceptable to have identical *Post* requests vary their response based on a number of factors, such as whether the information being sent was submitted previously.

7.2.2 HyperText Markup Language (HTML)

HyperText Markup Language (HTML) is the language used to describe web pages[3]. Markup languages surround data or information with tags in angular brackets < and > that describe either how the information should be presented or the information's content. Different markup languages include different sets of tags. HTML is strictly a presentation language. The tags state how things should be displayed. Below is a short HTML document that displays in Internet Explorer as shown in Figure 7.1.

[2] *Hypertext* documents contains links to other documents, that the reader can access, typically by clicking on the link with the mouse.

[3] Use of HTML is not limited to web pages. For example, it is also frequently used to format messages in email systems.

Figure 7.1: A simple web page

```
 1 <!--
 2 Hello.html
 3 A simple web page to illustrate html tags.
 4 Written by: Stuart Hansen
 5 Date: March 23, 2009
 6 -->
 7 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
 8 <html>
 9   <head>
10     <title>Hello World Page</title>
11     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
12   </head>
13   <body>
14     <h1>Hello World!</h1>
15   </body>
16 </html>
```

Lines	Commentary
1–6	Comments within HTML documents appear between `<!--` and `-->`.
7	This is the document type declaration. Including this declaration allows the document to be validated, guaranteeing that it is well-formed. The two primary families of document types currently in use for web pages are `HTML 4.01 XXX` and `XHTML 1.X XXX`. For our purposes the differences among the various versions may be ignored.
8–16	The part of the HTML document that will be displayed appears between the `<html>` and `</html>` tags. There are two required elements within the html document, <head> </head> and <body> </body>
9–12	The head element contains the title, meta-information, and the locations of scripts.
11	Meta information is information about the page. In this example, it tells us that the document is a text document encoded in UTF-8.
13–15	The body of the document contains the information that should be displayed in the web page. In this example, it contains one line, "Hello World!" that will be displayed as a top-level heading, <h1>.

There are many HTML tags and for our purposes it is foolish to try to memorize all of them. Instead, there are any of a number of software tools that will help produce nice looking web pages. As programmers[4] the authors recommend using NetBeans. NetBeans is bundled with a web server for displaying pages (see the next section), and drag and drop tools for inserting tags into pages.

7.2.3 Web and Application Servers

You probably already have a good notion of what a web server is. It is a program running on a computer that returns a web page when a request is made for it. Older web servers were designed just to return static web pages and not much more. As it became evident that dynamic web content was becoming increasingly important, web servers were extended to include the ability to run other programs to create the content. For example, an ability to execute Perl scripts was a standard add-on to early servers.

In recent years, as web applications have become more and more important, the industry has shifted from web servers to *application servers*. An *application server* is designed to expose the public interface of the application, efficiently creating dynamic content, while still being willing to respond with static pages when required. There are numerous application servers available from different vendors, including: Tomcat, GlassFish, JBoss, WebLogic, WebObjects and WebSphere.

Like static web pages, web applications need to be deployed to the server where they will run. Often this amounts to simply creating a subdirectory and copying files into it. Unfortunately, the details differ significantly from application server to application server. Fortunately, many servers now come with deployment tools to aid the developer.

7.3 Java Servlets

As in previous chapters, we will illustrate these concepts using Java.

Servlets are Java programs that run on an application server. The *servlet container* is the portion of the application server that executes the Java byte code. The servlet container is responsible for managing the servlets. For example, it maps URLs to servlets. It starts the Java virtual machine when a request is made for a servlet. It checks the access permissions of the requester to ascertain whether they have permission to run the servlet.

There are two files needed to work with servlets. On the browser side, we need a web page that requests that a servlet be run.

On the server side, we need the servlet. Servlets are pure Java code. They require a specialized Java library: `javax.servlet.*`. Our servlet class will extend `HTTPServlet`, overriding methods to accomplish our tasks.

7.3.1 Calculator Example

The ideas behind web applications and their dynamic content become much clearer when we look at an example. The examples in this chapter were developed using NetBeans 6.5. This IDE comes with graphical tools for building web pages and several application servers that can be used for development. It also comes packaged with the libraries (`javax.servlet.*`) that are needed to develop servlets.

Below is an abbreviated web page that contains a calculator. The web page is shown in Figure 7.2. The result of carrying out a calculator operation is shown in Figure 7.3.

[4]This is a programming text, after all.

```
10 <html>
11     <head>
12         <title>Calculator</title>
13         <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
14     </head>
15     <body>
16         <form name="Calculate" action="Calculate">
17             <input type="text" name="FirstNumber" value="0" />
18             <input type="text" name="SecondNumber" value="0" />
19             <select name="Operation">
20                 <option>+</option>
21                 <option>-</option>
22                 <option>*</option>
23                 <option>/</option>
24             </select>
25             <input type="submit" value="OK" name="SubmitButton" />
26         </form>
27     </body>
28 </html>
```

Lines	Commentary
1–9	The first 9 lines of the file contain heading comments and the data type declaration and are omitted to save space.
16–26	The body of the page contains a form, which is where the user enters information for processing. The `action=''Calculate''` attribute says that the `Calculate` servlet should be called when the form is submitted.
17–18	There are two text fields on the form, where the user enters the two numbers to be processed.
19–24	The form contains a drop down list, from where the user selects the operation to be performed.
25	The `Submit` button is clicked to send the request to the server.

There are more elements that can be included in a form, including radio buttons, multi-line input fields, and checkboxes. A complete discussion of html forms is beyond the scope of this text.

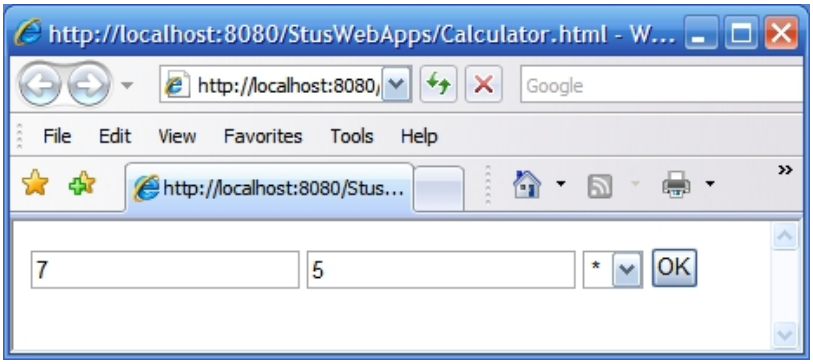

Figure 7.2: The interface for the calculator.

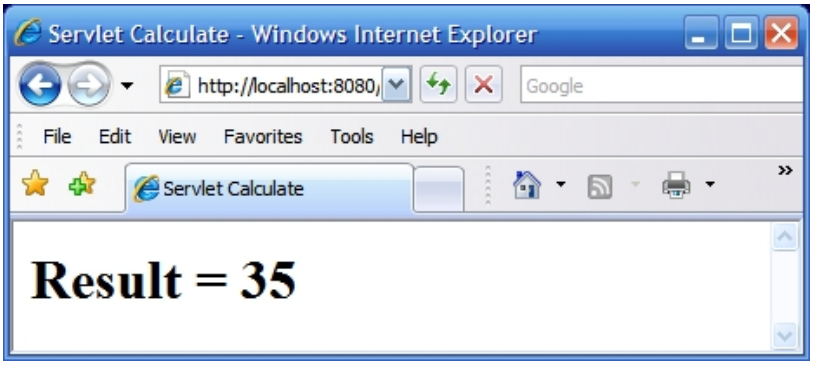

Figure 7.3: The result of carrying out a calculator operation.

The Calculate Servlet

```
 1 import java.io.IOException;
 2 import java.io.PrintWriter;
 3 import java.util.Enumeration;
 4 import javax.servlet.ServletException;
 5 import javax.servlet.http.HttpServlet;
 6 import javax.servlet.http.HttpServletRequest;
 7 import javax.servlet.http.HttpServletResponse;
 8
 9 /**
10  * Calculate.java shows how to process form data in a servlet.
11  * This file responds to requests generated by Calculator.html
12  *
13  * @author Stu Hansen
14  * @version March 29, 2009
15  */
16 public class Calculate extends HttpServlet {
17     /**
18      * Processes requests for both GET and POST methods.
19      * @param request servlet request
20      * @param response servlet response
21      * @throws ServletException if a servlet-specific error occurs
22      * @throws IOException if an I/O error occurs
23      */
24     protected void processRequest(HttpServletRequest request,
25         HttpServletResponse response) throws ServletException, IOException {
26         response.setContentType("text/html;charset=UTF-8");
27
28         // Get the three parameters passed in from the web page
29         int i = Integer.parseInt(request.getParameter("FirstNumber"));
30         int j = Integer.parseInt(request.getParameter("SecondNumber"));
31
32         // Carry out the requested operation
33         String op = request.getParameter("Operation");
34         int result = 0;
35         if (op.equals("+"))
36             result = i+j;
37         else if (op.equals("-"))
38             result = i-j;
39         else if (op.equals("*"))
40             result = i*j;
41         else if (op.equals("/"))
42             result = i/j;
```

```
43          // Respond back to the browser with the result
44          PrintWriter out = response.getWriter();
45          try {
46              out.println("<html>");
47              out.println("<head>");
48              out.println("    <title>Servlet Calculate</title>");
49              out.println("</head>");
50              out.println("<body>");
51              out.println("    <h1>Result = " + result + "</h1>");
52              out.println("</body>");
53              out.println("</html>");
54          } finally {
55              out.close();
56          }
57      }
```

Lines	Commentary
1–7	The servlet packages do not come with Sun's JDK, but may be downloaded separately. They do come bundled with NetBeans.
16	Servlet classes extend HttpServlet. This is an abstract class and programmer must override at least one method, generally `doGet()` and/or `doPost()`..
24 & 25	Both `doGet()` and `doPost()` call `processRequest()`. The `request` parameter contains all the data included with the request arriving from the web browser. The `response` parameter is used to dynamically generate a page and send it to the browser.
29, 30 & 33	We make three calls to `request.getParameter()` to obtain the two numbers and the operation we are to carry out. Note that all parameters are `String`s, so the numbers need to be massaged into their correct integer representations.
35–42	We use an `if..else if` structure to determine the operation and carry it out.
45–57	We dynamically generate a web page to be returned to the browser.

The remainder of the servlet is autogenerated by NetBeans. Both `doGet()` and `doPost()` call `processRequest()`, which is defined above.

```
61      /**
62       * Handles the HTTP <code>GET</code> method.
63       * @param request servlet request
64       * @param response servlet response
65       * @throws ServletException if a servlet-specific error occurs
66       * @throws IOException if an I/O error occurs
67       */
68      @Override
69      protected void doGet(HttpServletRequest request,
70                                          HttpServletResponse response)
71      throws ServletException, IOException {
72          processRequest(request, response);
73      }
74
75      /**
76       * Handles the HTTP <code>POST</code> method.
77       * @param request servlet request
78       * @param response servlet response
79       * @throws ServletException if a servlet-specific error occurs
80       * @throws IOException if an I/O error occurs
81       */
82      @Override
83      protected void doPost(HttpServletRequest request,
84                                           HttpServletResponse response)
85      throws ServletException, IOException {
86          processRequest(request, response);
87      }
88
89      /**
90       * Returns a short description of the servlet.
91       * @return a String containing servlet description
92       */
93      @Override
94      public String getServletInfo() {
95          return "Short description";
96      }
97  }
```

Deploying the Calculator Servlet

Deploying a web application is primarily the process of copying the application files to the appropriate subdirectory within the server's servlet container and then registering the servlet with the container. While this seems straightforward, the details get confusing[5]. NetBeans and other IDEs will autogenerate a `web.xml` script that `ant` will use to do the deployment. This is, by far, your best option for deploying web applications.

Running the Calculator Servlet

A servlet is invoked by having the browser send a request for the servlet to the server. In our case, the request will include parameters. Clicking the "Submit" button will generate the request, but

[5]Each servlet has a servlet name, a class name, and a URL pattern. Giving the same servket three different names strikes the authors as particularly confusing. We recommend that whenever possible use the same name for all three.

the user can also type the request into the browser's navigation bar. For the calculator, running on the local machine, on port 8080, the user would enter:

```
http://localhost:8080/StusWebApps/Calculate?FirstNumber=5&SecondNumber=7&Operation=*&SubmitButton=OK
```

7.4 Adding State to Web Applications

Recall that an important feature of event based systems is that they are state based. Unfortunately, HTTP was designed as a stateless protocol. That is, web servers are not required to retain any information about a client between the client's requests. This worked fine when the web consisted of static web pages, but has created havoc in a universe where we expect web servers to maintain shopping carts or other user specific data. For example, each time a user adds something to a shopping cart, the server should maintain that information, at least for a while. This is a change in the data state of the server.

It is also frequently necessary to maintain control state. Consider, for example, a two person game where the people alternate turns. A web interface for this game requires that the one player be able make a move and that the other player be blocked from moving. That is, there is control state needed.

The problem of maintaining state in web applications has been solved in a number of ways, including *cookies*, *sessions*, *hidden variables* and *databases*.

7.4.1 Cookies

A *cookie* is a small file sent by the server and stored on the client. The contents of the file will be returned to the server with subsequent requests. This method avoids problems with session timeouts, but creates several new issues. A server may store private information, e.g. credit card numbers, in a cookie, and others using the computer may see its contents. Cookies may track all the pages within a site that the user visits, building a profile of the user. This gives some people serious ethical concerns. Most browsers now allow the user to turn off cookies, preventing them from being stored.

7.4.2 Sessions

To understand *sessions*, we first need to understand a bit about a servlet's lifecycle. There are two practical considerations that guide this discussion. First, web and application servers handle many requests with limited resources. Thus, efficient management of resources is still very important on web servers. Second, when a user browses a website, they often visit the same or related pages repeatedly.

With limited memory resources, it makes sense for the servlet container to wait for a request for a servlet before loading it into memory. Once in memory, it makes sense to keep the servlet loaded for some time, as the user may well return to the servlet again. The period of time the user interacts with the servlet is known as a *session*. The servlet may remain in memory longer than that, as there may be multiple users interacting with the same servlet.

In modern web applications, there are many types of information that developers may want to keep for the duration of a session. For example, a shopping cart contains items the user is interested in purchasing, but hasn't yet paid for. We don't want these to be forgotten. Similarly, many websites now require logins. A session should keep track of whether the user has already logged in.

In most cases, it also makes sense to have sessions time out. That is, if a user makes no more requests of a servlet during a given time interval, the session should be dropped. The default timeout

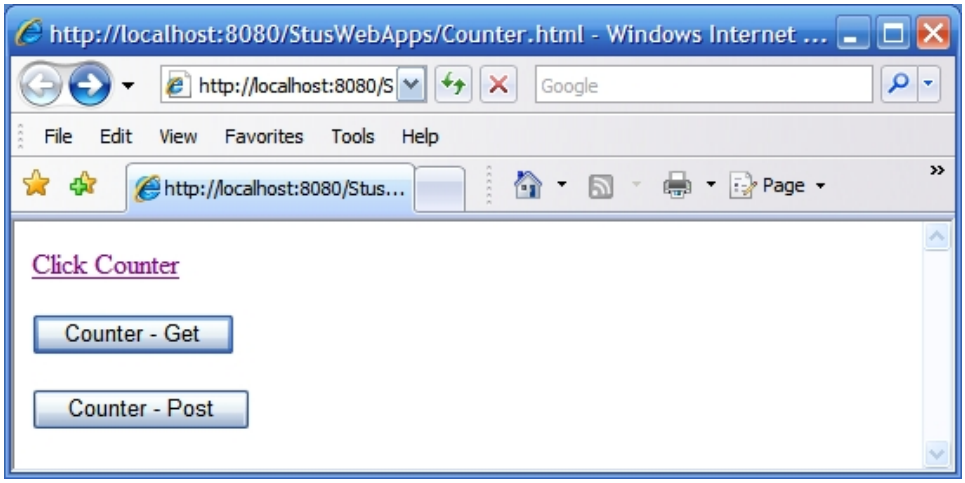

Figure 7.4: The interface for the counter application.

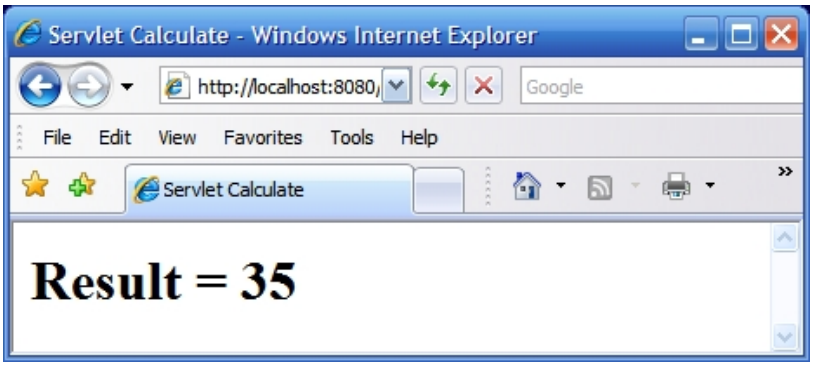

Figure 7.5: The page that results after the servlet has been visited 35 times.

period is typically 30 minutes. In Java, the `HTTPSession` class contains `setMaxInactiveInterval()` which allows the programmer to change that time.

Java's servlet library contains an `HttpSession` class. An `HttpSession` object is automatically created for a user's interactions with the servlet. The session object operates much like a hashtable, where the key is a string and the value is an arbitrary Java object. When we want to store state using a session, we obtain a reference to the session object and then get and set attributes within it.

The Counter Example

The following listing is for `Counter.html`, a web page with three ways to contact the same servlet: a link, a button that performs a get request, and a button that performs a post request.

```
 8 <html>
 9   <head>
10     <title></title>
11     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
12   </head>
13   <body>
14     <a href="Counter">Click Counter</a>
15     <form name="CounterGetForm" action="Counter">
16         <input type="submit" value="Counter - Get" name="GetCounter" />
17     </form>
18     <form name="CounterPostForm" action="Counter" method="POST">
19         <input type="submit" value="Counter - Post" name="PostCounter" />
20     </form>
21   </body>
22 </html>
```

Lines	Commentary
1–7	The first seven lines contain heading comments and data type declarations. They are omitted for brevity.
14	This link requests the servlet.
15	This button requests the servlet using "get".
18	This button requests the servlet using "post".

The excerpted listing below uses a session object to count how many times the servlet has been visited during this session. The entire code listing for this servlet is available on the text's website as `Counter.java`.

```
21      protected void processRequest(HttpServletRequest request,
22              HttpServletResponse response) throws ServletException, IOException {
23          PrintWriter out = response.getWriter();
24
25          HttpSession session = request.getSession(true);
26
27          int count = 0;
28
29          // Create the new session counter
30          if (!session.isNew()) {
31              count = ((Integer) session.getAttribute("count")).intValue();
32          }
33
34          // Increment the counter for this visit
35          count++;
36          session.setAttribute("count", count);
37
38          response.setContentType("text/html;charset=UTF-8");
39
40          try {
41              out.println("<html>");
42              out.println("<head>");
43              out.println("    <title>Servlet Counter</title>");
44              out.println("</head>");
45              out.println("<body>");
46              out.println("    <h1>Counter value for this session is: "
47                                                          + count + "</h1>");
48              out.println("</body>");
49              out.println("</html>");
50          } finally {
51              out.close();
52          }
53      }
```

Lines	Commentary
1–20	These lines contain imports, heading comments and the class declaration. They are virtually identical to the code in the previous example and are omitted for brevity. We only show the `processRequest()` method, which contains all the code of interest.
25	We obtain a reference to the `session` for this servlet. The `true` parameter directs the call to create a new session object i f one has not already been created. You may think of a session as a hashtable. It stores any object that the programmer wants to keep around between visits to the servlet.
30–32	In our case, we want to store the value of a counter that tells us how many times this client has visited this servlet during this session. Note that a client request arriving from another IP address will have its own session and counter.
36	Sessions use `String`s and keys and hold `Object`s. We use autoboxing to cast our int into an Integer. This relates back to line 31, where we we had to explicitly cast the `Object` to be an `Integer`, and then used auto unboxing to convert it to an `int`.
40–53	Respond with a dynamically created web page.
46	Note the use of `count`.

The remainder of the servlet code is autogenerated and is completely identical to that in the previous example, so is omitted.

7.4.3 Other Ways to Maintain State

Hidden Parameters

We have already seen how data can be collected from forms and sent to the server. *Hidden parameters* behave the same way as interface parameters, but are not visible on the web page. Web pages and servers can exchange information via hidden parameters.

Databases

Many web applications use database back-ends to store persistent information. For example, when you place an order for a cashmere sweater on the web, a database record will be created storing this information for further processing. Databases are very important in many web applications, but their role is to maintain information with very long lifespans (*persistent information*), much longer than session based information. The record of your sweater order will be around while the order is filled, shipped, arrived, and is possibly returned. This is much longer than a session.

7.4.4 Combining Request and Response Pages

The counter example in the previous section contains a rather large maintenance problem. The servlet is paired with a web page that requests it. A change to either will require a change to the other. Since the web page and servlet may be stored in different directories or on different machines, maintenance becomes an issue. If you consider doing a larger website, with numerous requesting pages and servlets, you can see how nightmarish the situation could become.

One solution to this problem is to make all the web content dynamic. Our requesting page was originally static. It was HTML code that was stored in a file. We requested that page, then used

that page to invoke the servlet which generated the counter response page. Instead, we can have a servlet generate both the requesting page and the response page. In our example, we can have the page be self contained. That is, we will put the counter and the controls to increment the counter on the same page. There is no reason why a servlet couldn't generate multiple very different pages, however, based on session or parameter values.

```
25      protected void processRequest(HttpServletRequest request,
26          HttpServletResponse response)   throws ServletException, IOException {
27          response.setContentType("text/html;charset=UTF-8");
28          PrintWriter out = response.getWriter();
29          try {
30              // Get the session variable
31              HttpSession session = request.getSession(true);
32
33              int count2 = 0;  // an integer counter
34
35              // Update the session's attribute, if it exists
36              if (session.getAttribute("count2") != null) {
37                  count2 = ((Integer) session.getAttribute("count2")).intValue();
38              }
39
40              // Increment the counter for this visit
41              count2++;
42
43              // Set the session's attribute
44              session.setAttribute("count2", count2);
45
46              // Return a new web page to the browser
47              response.setContentType("text/html;charset=UTF-8");
48              out.println("<html>");
49              out.println("<head>");
50              out.println("    <title>Servlet Counter</title>");
51              out.println("</head>");
52              out.println("<body>");
53
54              out.println("<h1>Counter value for this session is: "
55                                                              + count2 + "</h1>");
56              out.println("<form name=\"CounterGetForm\" action=\"Counter2\">");
57              out.println("    <input type=\"submit\" value=\"Counter - Get\"
58                                                          name=\"GetCounter\" />");
59              out.println("</form>");
60
61              out.println("</body>");
62              out.println("</html>");
63          } finally {
64              out.close();
65          }
66      }
```

The remainder of the servlet code is autogenerated and is completely identical to that of the previous example. It is omitted for brevity.

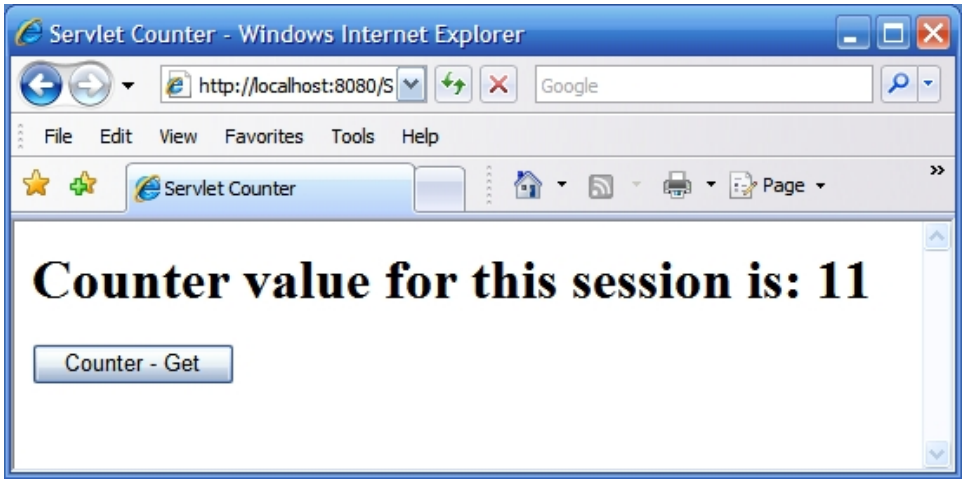

Figure 7.6: The interface for the self-contained counter servlet.

Lines	Commentary
1–24	These lines contain imports, heading comments and the class declaration. They are virtually identical to the code in the previous example and are omitted for brevity. We only show the `processRequest()` method, which contains all the code of interest. The entire code listing is available on the text's website.
27–44	These lines are virtually identical to those in the previous example. We get the `session` variable, see if there is a `counter2` attribute, and if not, initialize it.
46–61	Our `out.println`s dynamically create a web page.
54	Note the use of `counter2` on this line.
56–59	We include a form on the response. By doing so, we alleviate the need for a separate web page for input. Instead, the requesting page is just a different version of the response page.

7.5 Java Server Pages (JSPs)

Java Server Pages (JSPs) are an extension to servlets that make combining dynamic and static content a bit more natural. A JSP is identified by having the name suffix `.jsp`. A JSP looks much like a static web page, but has Java code embedded within it, between <% and %>. A servlet is autogenerated on the application server that wraps the JSP's content into something very much like the previous example.

The web page for this example is identical to the web page in the previous section, so its figure is omitted for brevity.

```
<%--
    Document   : JSPCounter
    This file contains a jsp page that keeps track of a counter
    Created on : Mar 29, 2009, 10:23:20 AM
    Author     : Stu Hansen
--%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Counter</title>
    </head>
    <body>
        <%
        // We include the code that is used to keep track of our counter
        // Note that jsp understand a number of default variables like:
        // request and session

        int jspCount = 0;  // Our counter

        // Check if this is the first time during this session that we
        // have been to the jsp.  If not, get the counter.
        if (session.getAttribute("jspCount") != null) {
            jspCount = ((Integer) session.getAttribute("jspCount")).intValue();
        }

        // Update the counter and store it back into the session
        jspCount++;
        session.setAttribute("jspCount", jspCount);

        // Respond to the browser with the counter
        out.println("<h1>Counter value for this session is: "
                                                    + jspCount +  "</h1>");
        %>

        <!-- In place of the out.println above, we could use the following -->
        <!--<h1> Counter value for this session is :
                  <% out.print(session.getAttribute("jspCount")); %> </h1> -->

        <form name="CounterGetForm" action="JSPCounter.jsp">
            <input type="submit" value="Counter - Get" name="JSPCounter" />
        </form>

    </body>
</html>
```

Lines	Commentary
1–10	These are heading comments and document type declarations. Note the difference in comment tags for JSPs rather than static web pages.
12–17	These are standard html tags.
17 & 37	The <% and %> tags surround a block of Java code that is embedded within the JSP.
18–36	This is standard Java code that looks very much like the code in `processRequest()` in the previous example. It is executed every time the JSP is loaded.
27, 33 & 36	JSPs automatically give the programmer access to certain standard variables, including `session` and `out`. There is no need to declare or instantiate them.
43–45	This again is standard html code that places the button on the page.
48–49	Terminate the open tags.

There is much more to JSPs than discussed here. There are custom tag libraries that make handling content easier. Students interested in delving further into JSPs are encouraged to pick up any of a number of excellent texts on the topic.

7.6 Web Services

The web application model presented in the previous chapter has been very successful, with many organizations deploying web applications as part of their web sites. How event based are they, however?

In some ways, web applications are quite event based:

- The web based front-end usually contains a form with text boxes and buttons, and code is associated with these components.
- The web application literature often refers to the web front-end as a view, the servlet as a controller, and the database as the model, again mirroring the terminology of the event based literature.

In other ways, web applications aren't all that event based:

- The web based front-end, the servlets running on the server, and the database back-end, although distributed, are all tightly coupled into one application. The association of the web pages, the servlets and the database is done statically, when the system is created, not dynamically, as we typically associate with events.
- The exchanges between the browser and server follow a request/response interaction, much like a method call. The browser requests a page from the server and sends along some parameters. The server returns the page. The interaction is asynchronous in the sense that the browser does not know how long it will take to receive a response from the server, but it is blocking in the sense that the browser waits for the response[6].

[6]Most browsers have a separate user thread that is not blocked, however, allowing the user to open another page or carry out other activities.

Web services utilize the same web infrastructure as web applications, but are built on a different model. A *web service* offers services to client programs. The *client* may be a web browser, a different web server, or a standalone program. The client requests a service from the hosting application server. The service response may be a web page, a different document, or nothing at all.

There are numerous examples of web services. Here are a few popular ones:

- *PayPal*, `https://developer.paypal.com/`, securely manages the financial portion of sales, so users no longer have to enter their credit card numbers to make on-line purchases. Both the consumer and the seller interact with PayPal to complete the transaction.

- *Google documents*, `http://docs.google.com`, lets users create, edit and store word processing documents, spreadsheets and presentations.

- *Blogger.com*, `http://www.blogger.com/developers/api/1_docs/`, lets programmers add blogging capabilities to their applications.

- *Flickr photo sharing*, `http://www.flickr.com/services/api/`, lets programmers add photo sharing capabilities to their applications.

Some authors argue that web services are the future of desktop computing. Applications and documents will be stored on servers. It won't matter where you are, or what computer you work at. As long as you have any computer and the Internet available, you will be able to access all your work.

Like many things in computing, web services continue to evolve at a rapid rate. The next few sections discuss the basic model used by web services. However, just about all aspects of this model are currently in a state of flux, with new ideas and technologies being introduced regularly. We will briefly address these issues at the end of the chapter.

7.6.1 Stock Quote Example

A good way to understand how web services work is to walk through an example.

Pretty Good Investment Advisers wants to provide a way for clients to look up stock quotes on-line. They develop a web service where the user enters a stock symbol. In real time, the service looks up the price of the stock on their corporate office's mainframe computer, and returns the result to the user. They advertise their new service at: `http://www.xmethods.net`. One of their best clients is a CS professor at a local college. She downloads the documents describing the stock quote service, then writes a client program to work with the service to look up stock prices. She runs her program and enters `IBM` as the stock symbol, The web service looks up IBM and returns the stock quote in a document very similar the one shown below. Her program parses this document and displays the result.

7.6.2 The eXtensible Markup Language (XML)

Web services rely on *eXtensible Markup Language (XML)* for several purposes. It is used to describe the services. It is used to request services, and it is used for the responses. Like HTML, XML uses text based tags to organize the content of documents. For example, below is a brief XML document describing a stock quote.

```
 1   <Stock>
 2      <Symbol>              IBM    </Symbol>
 3      <Last>              82.71    </Last>
 4      <Date>          12/15/2008   </Date>
 5      <Time>              1:00pm   </Time>
 6      <Change>            +0.51    </Change>
 7      <Open>              82.51    </Open>
 8      <High>              82.86    </High>
 9      <Low>               80.00    </Low>
10      <Volume>          3007015    </Volume>
11   </Stock>
```

Lines	Commentary
1 & 11	Each stock quote is between <Stock> and </Stock>. In our example we only have one quote, but it would be easy to extend to a list of stocks.
2–10	The stock quote consists of a variety of attributes, each within its own pair of tags. For example, the low selling price of IBM on December 15, 2008 was 80.00.

XML Languages, Schemas and SOAP

We need a standard way to represent stock quotations, otherwise the client programs will have a terrible time working with the service. The pattern for the standard can be seen in the IBM quote above. Each quote is surrounded by <Stock> and </Stock> and has a list of properties between. *An XML language* is a well defined set of XML tags that provides information on a particular topic, e.g. stock quotes, that follows a particular scheme. When we say we want a standard way to represent stock quotes, what we are saying in XML terms is that we want to define an XML Language.

An *XML Schema* describes an XML language. Thus, there is an XML StockQuote schema that describes all the tags that can occur in any stock quote document. Anyone implementing a stock quote service, or using a stock quote service can read the schema and what tags go into a stock quote, allowing them to create and parse stock quote documents[7].

The *Simple Object Access Protocol (SOAP)* is an XML language designed for communicating the structure and values of an object between systems. The request and response messages used by web services use SOAP to represent their content.

7.6.3 Finding a Web Service and its API

To be useful, a web service needs to advertise itself. This includes its locations (its URL) and its public interface (API). It does so by registering a *Web Services Description Language (WSDL)* document in a *Universal Description Discovery and Integration (UDDI)* registry. The basic model is shown in Figure 7.7. Note that the UDDI registry contains the URL for the WSDL document, not the actual document.

The *WSDL* document describes a service, including:

- the URL for the service,

[7] An XML schema is actually another XML document. Thinking a little recursively, there is also an XML schema that describes schemas.

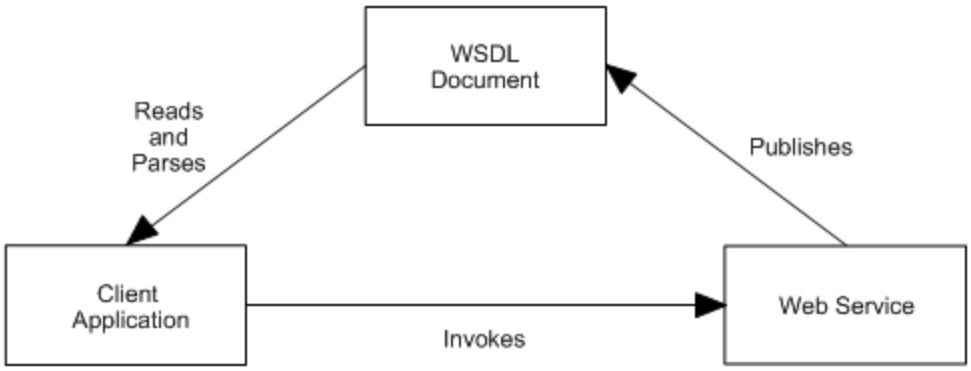

Figure 7.7: The basic pieces needed for a web service to be used.

- the public methods the service provides,
- arguments and types for the methods,
- the type of value returned from each method.

The UDDI registry entry contains:

- a reference (URL) to the WSDL document,
- information about the publisher of the service,
- a categorization of the service, and
- technical information about the service.

WSDL and UDDI are XML languages. Together the WSDL and UDDI documents for a service provide a web service client the information it needs to successfully access the service.

One popular UDDI registry is `http://www.xmethods.net`. When you browse this site, you will find thousands of web services. If you burrow down through a service's links, you will find complete descriptions of the service, including the WSDL document text.

When a programmer develops a web service client, they start by locating the WSDL documents for any services they will use. Most modern IDEs, including NetBeans and Visual Studio .Net, provide built-in methods to parse the WSDL documents and autogenerate code to make calls to the services methods. The programmer only has to write the client code that uses the service.

7.6.4 Summary of XML Uses in Web Services

We have thrown a lot of jargon and acronyms into this section, so here is a summary:

Term	Description
XML	eXtensible Markup Language: a generalization of html. Tags surround the content of a document. They are used to add meaning and formatting information. XML is used in numerous ways in web services.
WSDL	Web Service Description Language: the XML language for describing a web service. The wsdl document is used by a programmer to develop a web service's client.
UDDI	Universal Description Discovery and Integration: an XML language used to advertise a web service. UDDI registries contain UDDI documents for many different web services.
SOAP	Simple Object Access Protocol: an XML language used to represent objects. It is used for requests and responses in web services.
Schema	An XML document that describes an XML language. Thus, there is a schema for WSDL, UDDI, and SOAP. SOAP is also specialized for requests and responses in a particular web service. Each of these also has a schema.

7.7 Developing a Web Service

It is possible to code a web service or a client from scratch. The only reason to do so, however, is to thoroughly learn the underlying XML languages and protocols. Almost all popular languages have IDEs associated with them that will autogenerate code to interact with clients. This abstracts away the low-level details, allowing the programmer to concentrate on developing the service instead.

In this section we will develop a Java based web service using the NetBeans IDE. As briefly discussed in the previous chapter, NetBeans is a free IDE available from `http://www.netbeans.org`. It is packaged with several application servers making developing and deploying web services much simpler.

7.7.1 Quadratic Equations Revisited

Our example is that of a quadratic equation solving service. Recall from high school that quadratic equations have the form

$$ax^2 + bx + c = 0$$

`a`, `b`, and `c` are known as the coefficients. By changing `a`, `b`, and `c` we get different equations.

A solution to a quadratic equation is a value of `x` that makes the equality true. For example,

$$1x^2 + -3x + 2 = 0$$

has two solutions: $x = 1$ and $x = 2$ A quadratic equation may have 0, 1 or 2 solutions, depending on its coefficients.

You probably also recall that this type of equation may be solved using the quadratic formula, which is

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Our service will contain a single method, named `solve()`, which takes three parameters (`a`, `b`, and `c`). It will use the quadratic formula to find and return the solutions to the equation.

7.7.2 Initial Steps

The initial steps to creating the web service in NetBeans are:

1. Create a new Project.
   (a) In the dialogs choose `Java Web` and `Web Application`. Note that while we are not creating a web application as described in the previous chapter, we are using the same infrastructure, e.g. application servers and servlet classes.
   (b) Name the project anything appropriate.
   (c) Choose any installed application server. The authors recommend choosing `Glass Fish V2`, as it allows the programmer to interactively test the web service.
   (d) Click `Finish`.
2. Right click on the project icon and choose `New Web Service`. Name it `QuadraticService`. Name its package `QuadService`. This will generate the class stub shown below.

```
 1 package QuadService;
 2
 3 import javax.jws.WebService;
 4
 5 /**
 6  *
 7  * @author Stu Hansen
 8  */
 9 @WebService()
10 public class QuadraticService {
11
12 }
```

Lines	Commentary
3	Note the auto-generated `import` statement, that brings in the `WebService` interface.
9	The `@WebService()` annotation makes our class into a web service.

7.7.3 Web Service Annotations

Beginning with Java 1.5, the language has included annotations. Annotations are meta-data, describing some aspect of the code. An annotation begins with the `@` symbol and includes information and directives for tools processing the Java file.

Some annotations are used directly by the Java compiler. For example, `@Override` before a method name tells the compiler that this method is meant to override a method in a super class. If the signature of the method is incorrect, and it doesn't override a super class method, the compiler is required to generate an error. `@Override` is not required, nor does it affect the byte code generated. Instead, it is meant as a means of helping the programmer manage their code, guaranteeing that the signature of the method is correct.

Other annotations are used by different tools besides the compiler. For example, Sun's SDK includes `wsgen` and `wsimport`. These are command line tools to generate web service artifacts including classes and script files used to develop, deploy and invoke web services. Because we are

working within NetBeans, we will not call these tools directly. NetBeans will do it for us. The three annotations we will use that are recognized by these tools are: `@WebService`, `@WebMethod`, and `@WebParam`.

7.7.4 Completing the Web Service

We complete our web service by writing and annotating the `solve()` method. NetBeans has a GUI tool to help programmers define web methods. This is available by clicking the `Design` tab at the top of the editor pane. The annotations in our example are simple enough that they can easily be entered manually, too.

```
 1 package QuadService;
 2
 3 import javax.jws.WebMethod;
 4 import javax.jws.WebParam;
 5 import javax.jws.WebService;
 6
 7
 8 /**
 9  * A web service with exactly one method.
10  * The method solves quadratic equations.
11  *
12  * @author Stu Hansen
13  * @version April 2009
14  */
15 @WebService(name = "Quadratic", serviceName = "QuadraticService")
16 public class QuadraticService {
17     @WebMethod(operationName = "solve")
18     public double [] solve (@WebParam(name = "a") double a,
19                             @WebParam(name = "b") double b,
20                             @WebParam(name = "c") double c) {
21         double [] results = new double [2];
22
23         // Because Java's doubles use Infinity and Nan, we don't need to
24         // worry about exceptions.
25         double discriminant = b*b - 4*a*c;
26         results[0] = (-b + Math.sqrt(discriminant))/(2*a);
27         results[1] = (-b - Math.sqrt(discriminant))/(2*a);
28
29         return results;
30     }
31 }
```

Lines	Commentary
3–5	Our import statements need to be expanded to include all three types of annotations.
15	We expand the `@WebService()` annotation including `name` and `serviceName` parameters. These will be used later when developing the client in order to identify the service.
17	The `@WebMethod` annotation declares `solve()` to be a method that will be exposed in the web service's API. The `operationName` parameter just tells us that it will still be called `solve`.
18–30	The `solve()` method solves the quadratic equation using the quadratic formula. Because there may be up to two solutions, we return an array of doubles. Java's `double` uses `Infinity` and Not a Number (`Nan`) for divide by zero and taking the squareroot of a negative. Thus, no exception handling code is needed.
18–20	Each parameter is annotated with `@WebParam`. The name parameter for each tells us that each parameter will retain its own name (a, b, and c) in the web service.

7.7.5 Testing and Deploying the Web Service

As with any distributed computation, things become much more complex when we use multiple computers with a network joining them. As such, common sense tells us that we should test our web service locally, as much as possible, before deploying it. The `QuadraticService` class in our example, is just another Java class. The annotations do not affect our ability to test the code locally. We can (and should) write `JUnit` tests or a test `main()` to exercise the code removing any bugs discovered.

Deploying the web service is a matter of installing our web service class, along with autogenerated code onto an application server. NetBeans makes this easy. It has autogenerated an `Ant` deployment descriptor, `web.xml`. Right click on the project icon and choose `Deploy`. It may take several minutes to start the application server and deploy the web service, but it will all happen without further programmer intervention.

If you are using the `Glass Fish v2` application server, after the service is deployed, it can be tested interactively. In `NetBean`'s `Projects` window, expand the Web Services folder by clicking on the `+` sign. Right click on `QuadraticService` and choose `Test Web Service`. NetBeans will open a browser and display a page containing a form that lets you interactively enter `a`, `b`, and `c`. When you click the `solve` button, the browser sends a request to the web service, which returns the solution to the quadratic[8].

7.7.6 WSDL and Schema Revisited

How could NetBeans generate a web page to test the service? It must know where the service is located and its public API. This is exactly the information stored in the wsdl and schema documents for the service. These are quite long and complex, and we have chosen not to include them here. The key ideas to remember is that they allow the dynamic testing of the service, and shortly will be used to develop the web client.

[8] Way cool!

Request and Response Documents

The web service is incorporated into a servlet on the application server. Unlike the servlets in the previous chapter, however, the requests and responses to this servlet are XML documents. The form of these documents is specified in the web service schema.

Below is a sample request document for our web service:

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 3     <S:Header/>
 4     <S:Body>
 5         <ns2:solve xmlns:ns2="http://QuadService/">
 6             <a>1.0</a>
 7             <b>0.0</b>
 8             <c>-4.0</c>
 9         </ns2:solve>
10     </S:Body>
11 </S:Envelope>
```

Lines	Commentary
2	SOAP documents are surrounded by a SOAP envelope. Note that this tag tells us where the schema for SOAP documents is located.
4–10	This is the body of the request.
5 & 9	We are making a request to the `solve()` method.
6–8	The call to `solve()` takes three parameters, a, b, and c.

Here is the response document:

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
 3     <S:Body>
 4         <ns2:solveResponse xmlns:ns2="http://QuadService/">
 5             <return>2.0</return>
 6             <return>-2.0</return>
 7         </ns2:solveResponse>
 8     </S:Body>
 9 </S:Envelope>
```

Lines	Commentary
4–7	This is the body of the response to our request.
5–6	Two values are returned. 2.0 and -2.0 are the solutions to the quadratic equation sent in the request.

7.8 Developing a Web Service Client

One of the strengths of the web service model is that clients can be web browsers, other web servers, or standalone applications. In this section we develop a standalone client that consumes the quadratic web service.

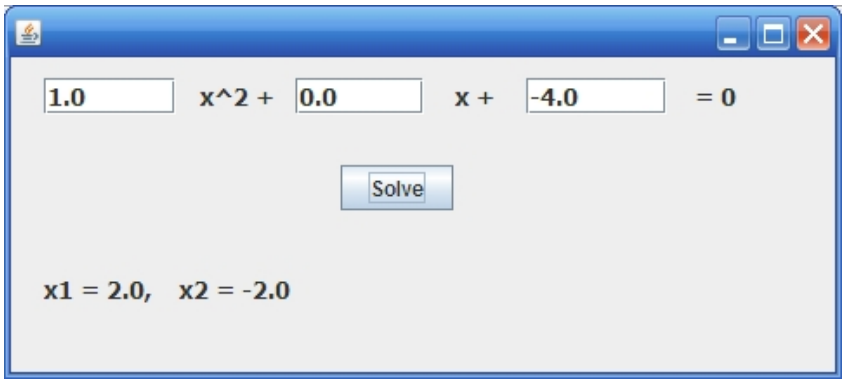

Figure 7.8: The GUI for the Quadratic Web Service Client.

7.8.1 Initial Steps

Figure 7.8 shows the GUI for the quadratic equation web service client. The user enters numbers for the coefficients of the equation and then clicks the Solve button. A message is sent to the web service, which responds with the solutions.

The initial steps to begin developing the client are:

1. Create a new NetBeans project. For this example, standard `Java Application` project is fine, although almost any type of project will work.
2. Import the web service wsdl. Make certain the web service has been deployed, then:
   (a) Right click on the project icon and choose `New -> Web Service Client`.
   (b) Since the web service is a NetBeans project developed on the same machine as the client, we leave the Project button checked.
   (c) Choose Browse. Navigate to `QuadraticWebService -> QuadraticService`.
   (d) Click OK, followed by Finish.
3. NetBeans will autogenerate proxy objects that will take care of all of the communication with the service. A new folder should appear in the project labeled: `Web Service References`. A number of Java files are generated. You don't need to do anything with these files, but you can see them by choosing the `Files` tab, then navigating to `build -> generated -> wsimport -> client -> quadservice`. The files include: `Quadratic.java`, `QuadraticService.java`, `Solve.java` and `SolveResponse.java`, as well as a couple of other support classes.

7.8.2 Completing the Client

Most of the code for the application is GUI code. Here we show only the snippets relevant to the web service exchange. The entire project is available for download from the text's website.

```
 2 import quadservice.Quadratic;
 3 import quadservice.QuadraticService;
```

Lines	Commentary
2–3	We import the two classes that act as the proxy for the service.

```
116     // All the work is done in the button's handler.
117     // We go to the service, solve the equation and display the results
118     private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
119     {
120         // We need a reference to the proxy (the Port) in order to call
121         // methods in the web service
122         QuadraticService quadService = new QuadraticService();
123         Quadratic quad = quadService.getQuadraticPort();
124
125         // Parse the coefficients
126         double a = Double.parseDouble(aField.getText());
127         double b = Double.parseDouble(bField.getText());
128         double c = Double.parseDouble(cField.getText());
129
130         // Find and display the solution.
131         // Note that the array of doubles in the service code has become a
132         // list in the proxy.
133         List<Double> solutions = quad.solve(a, b, c);
134         solutionsLabel.setText("x1 = " + solutions.get(0) +
135                                ",    x2 = " + solutions.get(1));
136     }
```

All of the interaction with the web service takes place in the button's handler.

Lines	Commentary
122 & 123	We obtain a reference to the web service by instantiating a `QuadraticService` object and then getting its port object. Note that these two names come from the `@WebService` annotation inserted in the web service class.
126–128	Convert the quadratic coefficients from Strings to doubles.
133	Call the `solve()` method on the web service via its proxy. Note that the web service class declared the return type to be double [ ], but here we use a List<Double>. This change of type was accomplished automatically, and was done because SOAP knows how to work with lists of objects, but not arrays of primitives.

7.9 Making Web Services More Event Based

Web services, as we have seen them so far, lack several of the properties that we have discussed as event based in earlier chapters. Since web services are servlets, they lack state, unless we use sessions or other special techniques. They also follow an asynchronous request-response interaction pattern.

While asynchronous request-response is event based, it can be argued that either a message passing system where no response is expected, or a publish-subscribe system is still more event based. For example, if no response is expected, then messages can easily be multicast, without waiting for any return messages.

7.9.1 Developing a Web Service with State

In this section, we show how to manage state in web services using session objects. Our example is a continuation of the quadratic equation problem. Rather than have the client provide the coefficients every time the service is called, however, we store the equations in session.

We only show code snippets for our web service in order to save space. As always, the complete code for the example is available on the text's website.

```
 3 import javax.annotation.Resource;
 4 import javax.jws.WebMethod;
 5 import javax.jws.WebParam;
 6 import javax.jws.WebService;
 7 import javax.servlet.http.HttpServletRequest;
 8 import javax.servlet.http.HttpSession;
 9 import javax.xml.ws.WebServiceContext;
10 import javax.xml.ws.handler.MessageContext;
```

Lines	Commentary
3–10	The imports are expanded to include several additional classes.
3	Resources are another type of Java annotation used to direct Java to autogenerate specific code. We will see it used below on line 23.
9 & 10	Two classes designed particularly to help us obtain and manage the session.

```
25 @WebService(name="QuadraticWithState", serviceName="QuadraticWithStateService")
26 public class QuadraticServiceWithState {
27
28     // Some variables to help create and manage a session
29     private @Resource WebServiceContext webServiceContext;
30     private MessageContext messageContext;
31     HttpSession session;
32
33     /**
34      * Web service operation to create an equation
35      */
36     @WebMethod(operationName = "create")
37     public int create(
38             @WebParam(name = "name") String name,
39             @WebParam(name = "a") double a,
40             @WebParam(name = "b") double b,
41             @WebParam(name = "c")  double c) {
42
43         // We only create a session if there isn't one already
44         if (session == null) {
45            messageContext = webServiceContext.getMessageContext();
46            session = ((HttpServletRequest) messageContext.get(
47                     MessageContext.SERVLET_REQUEST)).getSession();
48         }
49
50         Quadratic temp = new Quadratic(a, b, c);
51         session.setAttribute (name, temp);
52         //TODO write your implementation code here:
53         return 0;
54     }
55
56     /**
57      * Web service operation to solve the equation
58      */
59     @WebMethod(operationName = "solve")
60     public double [] solve(
61             @WebParam(name = "name") final String name) {
62         Quadratic temp = (Quadratic) session.getAttribute(name);
63         return temp.solve();
64     }
```

Lines	Commentary
23–25	Three instance variables to help us create and manage the session.
23	Note the use of `@Resource`. This directs Java to instantiate and make available the WebServiceContext object. Our code never instantiates it, but can use it whenever needed. It will appear in our code on line 39.
31–35	Our first method is now named `create()`. It takes four parameters.
32	name is used as the key when storing the equation in the session.
38–42	We check whether we have already created a session object. If not, we create a new one.
39	Here, we use `webServiceContext` to help us create a session.
44	`Quadratic` is a private inner class used to represent quadratic equations. They are the objects we store in the session.
45	Store the quadratic equation in the session.
53–58	`solve()` takes the name of the equation, looks it up in the session and solves the equation.
56	We get the equation from the session.

7.9.2 Client for a Web Service with State

Developing a client for this web service follows much the same pattern as in the earlier example. Here, we give a simple text based client.

```
 1 import java.util.List;
 2 import quadratic.QuadraticWithState;
 3 import quadratic.QuadraticWithStateService;
 4
 5 /**
 6  * A simple client that manipulates quadratic equations stored in a
 7  * web service.
 8  * @author Stu Hansen
 9  * @version April 2009
10  */
11 public class QuadraticClientWithState {
12     public static void main (String [] args)
13     {
14         // We use two objects to get to the service's methods.
15         QuadraticWithStateService qService = new QuadraticWithStateService();
16         QuadraticWithState quad = qService.getQuadraticWithStatePort();
17
18         // Create two quadratic equations
19         quad.create("Easy", 1.0, 0.0, -4.0);
20         quad.create("Tough", 1.0, 2.0, -6.0);
21
22         // Work with an easy equation
23         List<Double> solutions = quad.solve("Easy");
24         System.out.println("The solutions to: " + quad.toString("Easy")
25                                                             + " are: ");
26         System.out.println("x1 = " + solutions.get(0));
27         System.out.println("x2 = " + solutions.get(1));
28
29         // Work with a harder equation
30         solutions = quad.solve("Tough");
31         System.out.println("The solutions to: " + quad.toString("Tough")
32                                                             + " are: ");
33         System.out.println("x1 = " + solutions.get(0));
34         System.out.println("x2 = " + solutions.get(1));
35
36         // Just to show that the "Easy" equation is still there
37         System.out.print("Discriminant of " + quad.toString("Easy") + " is: ");
38         System.out.println(quad.getDiscriminant("Easy"));
39     }
40 }
```

Lines	Commentary
16 & 17	The web service reference and its port/proxy object.
20 & 21	Create two quadratic equations and store them in the web service.
23–37	Exercise various methods available in this web service. Note that `solve()` now takes the name of the equation, rather than the coefficients.

Here is the output from running the client program.

```
1 The solutions to: 1.0x^2 + 0.0x + -4.0 = 0 are:
2 x1 = 2.0
3 x2 = -2.0
4 The solutions to: 1.0x^2 + 2.0x + -6.0 = 0 are:
5 x1 = 1.6457513110645907
6 x2 = -3.6457513110645907
7 Discriminant of 1.0x^2 + 0.0x + -4.0 = 0 is: 16.0
```

@Oneway

Java web services also allow for `@Oneway` annotations. This annotation may be used with methods that have a `void` return type[9]. `@Oneway` is designed to improve efficiency, so that clients will not block waiting for a response from the server.

7.10 The Changing Landscape of Web Services

As stated earlier, the technologies and models used to develop web services continues to change. This section briefly presents some of the changes.

7.10.1 Web Services Inspection Language(WSIL)

UDDI registries never really caught on. In recent years, several of the larger ones have even shut down. One basic problem with them is that the client developer needs to refer to the UDDI document (located in a registry) and the WSDL document (located on a web server). The *Web Services Inspection Language (WSIL)* addresses this problem by combining the information found in the UDDI and WSDL documents into one document. WSIL documents can be stored anywhere. They don't need a special registry. They are generally found on the same machine as the web service. They can be shared publicly (if a service is trying to develop a clientele) or privately (if a service is only for already established service partners).

7.10.2 SOAP versus JSON

Because they are text, SOAP documents are readable by both humans and computers. To enhance human readability, we want each tag pair to be descriptive. For example, if we are describing a person, we might very well have the tag pair, <firstName> and </firstName>. This makes for verbose documents, however, as "firstName" is spelled out completely twice. If a network connection is slow, or there is a lot of network traffic, a long SOAP document will be slow to transmit.

[9]While Java's web services do not include OUT or INOUT parameters, other languages support this feature, and there are ways to make Java recognize them. `@Oneway` is also disallowed if these are used.

JavaScript Object Notation (JSON) is an alternative to SOAP. It is a light weight data-interchange format. JSON describes objects using name value pairs. The stock quote document above might be represented in JSON as:

```
1  {
2      "Stock": {
3          "Symbol": "IBM",
4          "Last":   82.71,
5          "Date":   "12/15/2008",
6          "Time":   "1:00pm",
7          "Change": 0.51,
8          "Open":   82.51,
9          "High":   82.86,
10         "Low":    80.00,
11         "Volume": 3007015,
12     }
13 }
```

The code is shorter and possibly more readable.

JSON based web services have been becoming more popular in recent years.

7.10.3 RESTful Web Services

Roy Fielding, one of the principal authors of the HTTP specifications, developed the notion of *Representational State Transfer (REST)* as an architectural style for hypermedia systems, including the World Wide Web. REST is not a standard, but rather a set of principals for designing web services. The key idea is to map resources to unique URIs. For example, each method in a web service has its own URI. In the Java world, this is equivalent to saying that each method gets its own servlet. By contrast, in a SOAP based system, the method name is part of the SOAP document. The web service parses the document and then calls the method. There is a single servlet that does the parsing and dispatching for all the methods.

There are several advantages claimed by RESTful web services:

- does not require a separate discovery mechanism, as all operations are URIs,
- may be more efficient on the server side, as results can be cached,
- depends less on vendor supplied software, as there is not need for additional messaging layers, and
- provides better long term ability to evolve and change than other approaches.

7.10.4 Evolving Language Tools

A final problem to discuss is the changing set of tools available for developing web services and clients. Using Java as an example, the early package for web services was JAX-RPC. This was replaced with JAX-WS (WS stands for web services), which is not backwardly compatible with JAX-RPC. Recently, with the growing popularity of RESTful web services, Java has come out with JAX-RS. The situation is no better in other languages. Computer science students need to accept that there will be ongoing demands to upgrade their knowledge and skills throughout their careers. Web services is just one example of where this problem is currently a dominant theme.

7.11 Conclusion

Web applications are closely related to the event based paradigm. The web based front end generally contains GUI elements similar to those found in standalone GUI apps. The communication with the web server is asynchronous, because the browser does not know how long a request/response round trip will take.

In other ways, web applications are quite far from the event based paradigm. The request/response interaction with the server is blocking on the browser's side. That is, after the browser requests a new page, the browser blocks other user interactions until the page is loaded. The association of handlers (servlet code) with particular requests is done statically. The best we can do is to use request parameters to change the code that is executed via a switch statement, just as we did in the Calculator example.

Web services are more event based. Since there may be many types of clients for a web service, it is strictly up to the client developer to decide whether the client blocks of not. For some messages sent to the server, no response is even required.

Web applications and services are still new. The technologies keep changing at an alarming fast rate. The concepts of event based programming will continue to apply, however, regardless of the technology that is currently hot.

Bibliography

[Angel, 2008] Angel, E. (2008). *Interactive Computer Graphics: A Top-Down Approach Using OpenGL, 5/E.* Prentice-Hall.

[Beck and Andres, 1999] Beck, K. and Andres, C. (1999). *Extreme Programming Explained: Embrace Change, 2nd ed.* Addison-Wesley Professional.

[Beer and Heindl, 2007] Beer, A. and Heindl, M. (2007). Issues in testing dependable event-based systems at a systems integration company. *Proceedings of the The Second International Conference on Availability, Reliability and Security*, pages 1093–1100.

[Deitel and Deitel, 2007] Deitel, H. M. and Deitel, P. J. (2007). *Java: How to Program, 7/E.* Prentice-Hall.

[Deitel et al., 2002] Deitel, H. M., Deitel, P. J., and Santry, S. E. (2002). *Advanced Java 2 Platform: How to Program.* Prentice-Hall.

[Elliott et al., 2002] Elliott, J., Eckstein, R., Loy, M., and Wood, D. (2002). *Java Swing, Second Edition.* O'Reilly.

[Englander, 1997] Englander, R. (1997). *Developing Java Beans.* O'Reilly.

[Food and Drug Adminstration, 2006] Food and Drug Adminstration (2006). Fda press release (august 28, 2006): United states marshals seize defective infusion pumps made by alaris products–pumps can deliver excess medication and harm patients. Online. Internet. Available WWW:http://www.pritzkerlaw.com/alaris-infusion-pump-signature-edition/index.htm#pressrelease.

[Gamma et al., 2000] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (2000). *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

[Hyde, 2003] Hyde, R. (2003). *The Art of Assembly Language.* No Starch Press.

[Johnson, 2000] Johnson, J. (2000). *GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers.* Morgan Kaufmann.

[Lamport, 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21.

[Muhl et al., 1998] Muhl, G., Fiege, L., and Pietzuch, P. R. (1998). *Distributed Event-Based Systems.* Springer.

[Norris McWhirter, 1985] Norris McWhirter, e. (1985). *The Guiness Book of World Records, 23rd US edition.* Sterling Publishing Co., Inc.

[Shreiner et al., 2007] Shreiner, D., Woo, M., Neider, J., and Davis, T. (2007). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2.1, 6/E.* Prentice Hall.

[Stallings, 2008] Stallings, W. (2008). *Operating Systems: Internals and Design Principles, 6/E.* Prentice Hall.

[Tanenbaum, 2007] Tanenbaum, A. S. (2007). *Modern Operating Systems, 3/E.* Prentice Hall.

[Trewin and Pain, 1998] Trewin, S. and Pain, H. (1998). A model of keyboard configuration requirements. In *Assets '98: Proceedings of the third international ACM conference on Assistive technologies*, pages 173–181, New York, NY, USA. ACM.

[Walrath et al., 2004] Walrath, K., Campione, M., Huml, A., and Zakhour, S. (2004). *The JFC Swing Tutorial: A Guide to Constructing GUIs (2nd Edition).* Addison-Wesley.