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.