Event-Based Programming

Taking Events to the Limit

Ted Faison

Apress®

Event-Based Programming: Taking Events to the Limit

ISBN-13: 978-1-59059-643-2

ISBN-10: 1-59059-643-9

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit www.apress.com.

The source code for this book is available to readers at www.apress.com in the Source Code section.

CHAPTER 2

Events and Notifications

Talking about events can sometimes be confusing, because people use the word *event* in different ways in different software contexts. For example, graphical user interface (GUI) operating systems use *events* to interact with applications. Applications must be written to conform to an *event-driven* model, doing nothing until an event occurs. When the OS determines that an event of interest to an application has occurred, the OS notifies the program. In this context, the information is sent using procedure calls, with additional parameters telling the application what type of event occurred, such as movement of the mouse or the pressing of a key on the keyboard. The application is free to take any action it wants in response to the event *notification*—including nothing.

Another use of the word *event* is in the context of concurrent programming. Programs set and reset *events*, which are process synchronization primitives such as semaphores and mutual exclusions (mutexes). Events are basically flags that one process can set and another can read.

A popular OO design pattern, called the *subject-observer* pattern, is based on events. The pattern is used when an object, called the *observer*, is interested in knowing about events that occur in another object, called the *subject*. The observer tells the subject which events it is interested in. When the subject detects one of these events, it sends an event notification to the observer, often by calling one of the observer's methods. A subject can theoretically have any number of observers.

Programming languages also use the word *event* to denote actions or entities of various kinds. In .NET components, events are associated with *delegates*, *event variables*, and *handlers*. In Java, events are associated with *event sources*, *event objects*, and *event listeners*. When used alone in a Java context, the word *event* usually indicates the object passed as an argument from an event source to an event listener. You can organize these events into a hierarchy with specialized event types derived from general ones, forming an event *type system*.

Defining Events and Notifications

In ordinary language, an event is an occurrence of some kind, while a notification is a message informing the recipient that something (presumably important) has happened. In the software context, the definitions of events and notifications are inextricably bound, with one being defined in terms of the other. Events are a cause; notifications are an effect.

Definition An event is a detectable condition that can trigger a notification.

Definition A notification is an event-triggered signal sent to a run-time–defined recipient.

The *condition* that is associated with an event might be based on any number of predicates—for example, that today is Tuesday, that the time is 10:30 a.m., or that my cat licked his left front paw for the seventh time in the last five minutes. The detection of the condition must be associated with a notification. Software systems detect all sorts of conditions. Of all those conditions, the only ones I'll call *events* are those that can trigger a notification.

Let's analyze a bit more what is and is not in the definitions. For starters, a condition must be detectable to be called an event. For example, consider the following C# code snippet:

```
void DoSomething()
{
  for (int x = 0; x < 10; i++)
    Console.Out.Write("hello");
}
```

The loop prints a string 10 times to the output console. For each iteration, the control variable x has a different value. According to the definition of event, can you consider the condition in which x assumes the value 3 to be an event? No, because the condition is not detectable in the code. When x assumes the value 3, nothing special happens. Let's add some code to detect the condition:

```
void DoSomething()
{
  for (int x = 0; x < 10; x++)
  {
    if (x == 3)
      Console.Out.Write("hello");
  }
}
```

Is the condition an event now? The answer is still no. Although the condition is detectable, no notification mechanism is associated with the condition. The code prints a string, but it doesn't use a notification to do so. The call to `Console.Out.Write` is not a notification, because the callee is identified at compile time. The caller always knows who the callee is. Let's change the code again:

```
public delegate void Updater(string theMessage);
public Updater xHasValueThreeHandler;

void DoSomething()
{
  for (int x = 0; x < 10; i++)
  {
    if (x == 3)
      if (xHasValueThreeHandler != null)
        xHasValueThreeHandler ("Third iteration");
  }
}
```

Now you can say that the condition (in which x assumes the value 3) is an event. The previous code introduced a new *delegate* type named `Updater`. In C#, delegates are basically pointers to methods. The code uses the type `Updater` to define a field called `xHasValueThreeHandler`. You can initialize this field at run time to point to any method of any class, as long as the method accepts a single string parameter and returns void. At compile time, there's no way to know if `xHasValueThreeHandler` will point to a method or not, so you must test the field before using it to ensure that it has been initialized to point to something. When the event is detected, a notification is sent in the form of a call to the method pointed at by `xHasValueThreeHandler`, if such a method exists.

Let's look at notifications in more detail. A notification is essentially a signal sent when an event occurs. The signal can be sent in two basic ways: by transferring data or by transferring execution control.

When the signal transfers data, the signal is a message written to a resource that is shared by the sender and receiver. The shared resource might be a network connection, shared memory, an OS service, a pipe, a computer, or something else. The message can be anything from a single value to a formatted message with fields describing the event. The fields might include a sender identifier, a message identifier, a time stamp, a priority value, or other information related to the event. In order for the notification to have an effect, the receiver must monitor the shared resource and be able to detect any new data written to it.

When the signal transfers execution control, the signal is sent as a procedure call. Depending on the relative locations of the sender and receiver, the procedure call might be local or remote. The signal is the call itself, with optional data passed as parameters or returned values. Figure 2-1 shows the steps involved in the production of a notification:

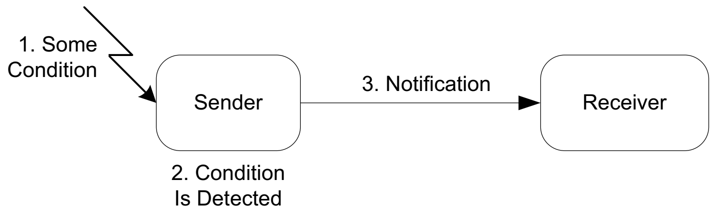

Figure 2-1. *Notifications as signals sent from one place to another*

The receiver doesn't necessarily have to do anything when it receives a notification. What it does is not the concern of the sender. In Figure 2-1, the arrow shows the direction of travel of the signal. Although the arrow *usually* shows the direction of data flow, this isn't always the case. For example, if a notification is delivered with a procedure call, the called method might have no input parameters and return a value. In this case, the direction of data flow would be in the opposite direction of the notification.

Notifications can be sent pretty much anywhere, limited only by their implementation strategy. Notifications can cross process boundaries, machine boundaries, and even network boundaries. Notifications sent by one component sometimes trigger events in others, producing a sequence of notifications and events that propagate through a system in a chain reaction. One component's notifications are another component's events, as shown in Figure 2-2.

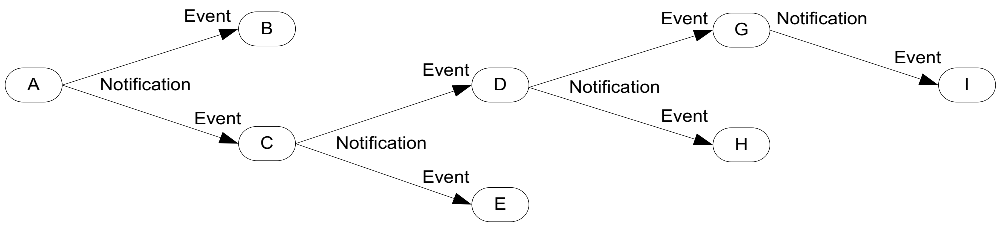

Figure 2-2. *Notifications and events in a chain reaction*

OO languages often provide built-in support for notifications based on procedure calls. In C#, VB .NET, and Delphi Object Pascal, the procedure to call is known as the *event handler*. The sender stores a reference to the event handler in a variable. As mentioned earlier, .NET languages such as C# and VB .NET use a new built-in type called a *delegate* to hold the references to event handlers. These delegates can store references to multiple event handlers. Object Pascal uses a simple pointer that can only reference one event handler at a time.

In .NET languages and Object Pascal, the sender calls the event handler without using a class or interface reference. Receivers are not required to implement any specific interface; therefore, receivers are called *untyped*. In contrast, in the JavaBeans event model, event listeners are *typed*. To receive a certain type of event, a listener must implement a specific interface. The sender uses a reference to this interface to call one of the methods of the receiver. When notification delivery is based on typed receivers, type coupling is introduced between the sender and the receiver. The sender not only identifies the interface of the callee, but it also chooses which method of that interface to call. Both decisions must be made at compile time, but the identity of the object called isn't determined until run time, when the sender's reference-to-interface is initialized.

A Brief History of Events

Events have been around for quite some time in software systems, although under different names. I'll summarize some of the most important developments.

Distributed systems started using event-oriented models in the 1970s. The first systems used shared memory, starting with a centralized shared memory model in multiprocessing systems.[1] The memory represented the rendezvous area for senders and receivers. Processors communicated by posting messages to the shared memory. Distributed shared memory models, with systems interconnected by network connections,[2] provided better performance and resilience to failures.

The Model-View-Controller design pattern[3] developed in the Smalltalk world is probably the first paradigm to display a clear event-based architecture. Events were used to maintain synchronization between the data (model) of a system and observers (views) of the data. The relationship between the model and views was later generalized as the subject-observer design pattern,[4] popularized in the 1990s.

Researchers in the artificial intelligence (AI) community used the shared memory model to support the *blackboard* paradigm.[5] AI processes collaborated in a multiprocessing system to interpret data. Some processes acted as producers of information and would post messages on the blackboard. Other processes would keep an eye on the blackboard contents, taking any messages that were related to their field of expertise.

In the mid-1980s, the Actors system[6] became one of the first multiprocessing systems to use events in the modern sense. Concurrent processes interacted using a pattern-oriented broadcasting

1. W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "HYDRA: The Kernel of a Multiprocessor Operating System," *Communications of the ACM*, June 1974.
   R.J. Swan, A. Bechtolsheim, Kwok-Woon Lai, and John Ousterhout, "The Implementation of the Cm* Multi-Microprocessor" (proceedings of the National Computer Conference, Dallas, TX, June 1977).
2. Brett D. Fleisch, "Distributed Shared Memory in a Loosely Coupled Distributed System," *Proceedings of the ACM Workshop on Frontiers in Computer Communications Technology* (New York: ACM Press, 1987).
3. Glenn E. Krasner and Stephen T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *Journal of Object-Oriented Programming*, August/September 1988.
4. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Boston: Addison-Wesley Professional, 1995).
5. Barbara Hayes-Roth, "A Blackboard Architecture for Control," *Artificial Intelligence*, July 1985.
   Robert Engelmore and Tony Morgan, *Blackboard Systems* (Boston: Addison-Wesley Professional, 1988).
   Vasudevan Jagannathan, Rajendra Dodhiawala, and Lawrence S. Baum, *Blackboard Architectures and Applications (Perspectives in Artificial Intelligence)* (Burlington, MA: Academic Press, 1989).
6. Gul Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems* (Cambridge, MA: The MIT Press, 1986).

mechanism, based on late-bound procedure calls. Around the same time, a number of rule-based systems were developed, such as Darwin.[7] These systems were based on communicating objects that used messages in an event-oriented manner.

In the meantime, starting in the late 1970s, module interconnection languages (MILs) gained favor. The first one was MIL75.[8] MILs seek to describe the modules that make up a system, using a formal grammar. MILs describe the interfaces of modules and the connections between modules, without regard to how the modules are implemented internally. Work on MILs continued into the 1990s. Polylith[9] used a MIL to specify connections between inputs and outputs. Polylith employed a *software bus* as an interconnection and integration medium. ToolBus[10] is a more recent example of a system using a software bus. Components known as *tools* are connected to the ToolBus, used as a multiprocessing message-dispatching infrastructure. Tools interact by sending and receiving messages over the ToolBus.

GUIs also have incorporated events in various degrees and fashions. In the early 1980s, GUI operating systems started taking hold, first with the Apple Macintosh, then with Microsoft Windows and MIT's open source X Window System. Applications developed for these operating systems are event-driven. Rather than containing their own input message-processing loop, applications are *reactive*, doing basically nothing until the operating system notifies them of input by sending event messages. All events are sent to an application's common event handler. Events include an identifier, telling the application what event occurred, plus optional data about the event.

In the late 1980s, a number of people worked on software integration systems, which were essentially frameworks for connecting modules or programs together. The Field system[11] was one of them. It used string-based messages as events to integrate windows and other tools together to implement a multiwindowed software development environment. The system was based on a centralized message subscription/notification facility. Tools, such as a source-code editor, could subscribe to certain types of messages, such as mouse messages. Notifications could be both synchronous and asynchronous.

In the 1990s, Architecture Description Languages (ADLs) became popular. ADLs can be viewed as an incarnation of MILs, but they're oriented more toward components and their connections. Examples of ADLs are Rapide,[12] Wright,[13] C2,[14] and ACME.[15] Many ADLs deal directly with events, considering them first-class citizens. For example, Rapide supports the concept of sets of causal events, in which each event can provoke one or more events in a set. Rapide uses the term *poset* (for *partially ordered set*) to identify a group of related events.

---

7. Naftaly H. Minsky and David Rozenshtein, "A Software Development Environment for Law-Governed Systems," *ACM SIGPLAN Notices*, February 1989.
8. Frank DeRemer and Hans Kron, "Programming-in-the-Large Versus Programming-in-the-Small," *IEEE Transactions on Software Engineering*, June 1976.
9. James M. Purtilo, "The Polylith Software Bus," *ACM Transactions on Programming Languages and Systems*, January 1994.
10. Jan A. Bergstra and Paul Klint, "The Discrete Time ToolBus: A Software Coordination Architecture," *Science of Computer Programming*, July 1998.
11. Steven P. Reiss, "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software*, July/August 1990.
12. David C. Luckham and James Vera, "An Event-Based Architecture Definition Language," *IEEE Transactions on Software Engineering*, September 1995.
13. Robert Allen and David Garlan, "Formalizing Architectural Connection" (proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 1994).
14. Nenad Medvidovic, Richard N. Taylor, and E. James Whitehead, Jr., "Formal Modeling of Software Architectures at Multiple Levels of Abstraction," (proceedings of the California Software Symposium, Los Angeles, CA, April 1996).
15. David Garlan, Robert Monroe, and David Wile, "ACME: An Architectural Interconnection Language" (technical report, Carnegie Mellon University, Pittsburgh, November 1995).

The 1990s marked the period in which major software vendors started marketing event-oriented programming languages. In 1991, Microsoft launched Visual Basic (VB), an important milestone in the history of event-oriented languages. It was the first widely used language with built-in support for events, which were based on untyped procedure calls. VB events were initially limited to GUI controls, such as forms, radio buttons, and checkboxes. VB defined a set of events for each type of control. The system made it very easy for programmers to add event handlers. Although somewhat limited, VB introduced the concept of event-based programming to millions of people and dramatically simplified the task of building user interfaces for the Windows platform. Borland took the event idea from VB and developed a Pascal version, called Delphi. Borland had made its fortune with a product called Turbo Pascal, and the company developed an object-oriented Pascal for the Delphi product. The new Object Pascal language, like VB, also uses procedure calls to deliver events.

Also in the 1990s, distributed component models became popular. The first was the Object Management Group's (OMG's) Common Object Request Broker Architecture (CORBA) in 1991. Some years later, the CORBA Event Service and CORBA Notification Service specifications were added.[16] CORBA events are based on the notion of *event channels*, which carry notifications from a source to a target, both of which are CORBA objects. CORBA supports asynchronous interactions, with provisions for load balancing and recovery. Notification payloads are described using OMG interface definition language (IDL). All CORBA specifications are given in terms of abstract interfaces. Implementation details are left to vendors.

Microsoft introduced Distributed Component Object Model (DCOM) in the mid-1990s. DCOM was an attempt to scale the Component Object Model (COM) architecture to distributed systems, but has since been replaced by the .NET Framework. The .NET Framework is an ambitious enterprise by Microsoft to create a cross-language run-time environment for local and distributed systems. The .NET Framework is a component-based technology that overcomes many of the limitations of Microsoft's COM and DCOM architectures, which developers consider too complicated. The .NET Framework includes support for events, remote procedure calls, and messaging services.

Another significant component model is Sun's JavaBeans. Initially a model for local systems, it was extended into an environment, known as Java 2 Platform, Enterprise Edition (J2EE), to support distributed systems. J2EE includes support for transactions as well as message queuing through the Java Message Service (JMS) API.

During the 1990s, a number of systems were proposed to overcome limitations of the three major component models (CORBA, JavaBeans, and COM). One limitation was incompatibility between them. Components built with one model couldn't interoperate with other models. Several bridging frameworks arose, in the form of model-to-model bridges, such as Iona's COM-to-CORBA bridge COMet.[17] Integration frameworks go beyond the bridging idea, attempting to create a homogeneous environment for disparate component models. For example, the Vienna Component Framework[18] is an environment that uses special *façade* components to support the interoperability of the three major models.

The three major models had several other shortcomings, including lack of support for rapid application development (RAD). Espresso[19] was a Java-based system designed to support tool-based visual composition and Internet distribution. Espresso components interacted exclusively using event notifications. Components could be wired together in an integrated development environment

16. CORBA Event Service Specification, www.omg.org/technology/documents/formal/event_service.htm, 2001.
17. Ronan Geraghty, Sean Joyce, Tom Moriarty, and Gary Noone, *COM-CORBA Interoperability* (Upper Saddle River, NJ: Prentice Hall, 1998).
18. Johann Oberleitner, Thomas Gschwind, and Mehdi Jazayeri, "The Vienna Component Framework Enabling Composition Across Component Models" (proceedings of the 25th International Conference on Software Engineering, Portland, OR, May 2003).
19. Ted Faison, "Interactive Component-Based Software Development with Espresso" (technical report, www.faisoncomputing.com/espresso/Introduction.pdf, 1997).

supporting visualization of a system at various levels of abstraction—from the contents of a single component all the way up to the layout of an entire system.

Messaging services, also known as message queues or store-and-forward systems, are used in distributed systems for the reliable delivery of messages, which can be considered in many ways as notifications. Messaging services started receiving widespread interest in the mid-1990s. Messaging services use a central message queuing server[20] that acts as a notification server for messages in a distributed system. Several commercial queues are available, including IBM WebSphere MQ (formerly MQSeries), TIBCO Rendezvous, Microsoft Message Queuing (MSMQ), and others. In message queuing systems, client programs send messages to the queuing server, with information regarding the intended recipients and (optionally) the quality of service (QoS) desired. The QoS might specify things such as support for transactions and delivery of return receipts. Messages are persisted locally on the message queue server, typically in a database or in disk files. Delivery is attempted according to a message priority scheme. If the addressee is offline or not available, delivery is attempted repeatedly until a limit is reached, such as a timeout.

The history of events and event-based systems (EBSs) is very rich and includes a large number of people, papers, and products. Due to the limited space available in this book, I have described only the most significant ones. My apologies to the people and projects that I was forced to omit. For another look at the history of event-based systems and current trends, I recommend reading the paper by Eugster.[21]

Nomenclature and Semantics

Before getting too wrapped up in details regarding events, I need to introduce some of the terminology used in EBSs. Distributed systems tend to use different terms and expressions than local (nondistributed) systems, and all the major event-oriented component infrastructures use their own expressions.

Let's start with the terminology used in this book. As mentioned earlier, an event is the detection of a condition, triggering a signal that is sent to interested parties. The entity that is able to detect events is called the *event publisher,* the *event source*, or simply the *sender*. The entity that receives notifications is called the *event subscriber*, the *event handler*, the *notification target*, the *notification receiver*, or simply the *receiver*. The act of sending the notification signal is also called *sending a notification* or *firing the event*. Any data passed with the notification is called the notification *payload*. Before an event can be fired, there must be receivers. The act of establishing a receiver is called *subscription* or *registration*.

Distributed systems use different terms and expressions. As mentioned earlier, EBSs are also called *publish-subscribe* systems. Publishers are components or objects that can detect events and send notifications. Subscribers are components or objects that receive notifications. In order for a subscriber to receive notifications, the subscriber must register by *subscribing* to the publisher. When dealing with publish-subscribe systems that include middleware systems, I'll sometimes use the term *event* to refer to incoming notifications from upstream, because many middleware systems consider incoming signals as events.

An orthogonal nomenclature is also in widespread use in distributed systems that use message servers. These systems often go by the names *message-oriented middleware* (MOM), *store-and-forward systems*, *message-based systems*, and others. Notifications are called *messages*, because they often consist of pure data packets sent over a network connection. Instead of *firing events*, these systems *send messages*. Entities that register to interact with the server are called *clients*. Clients can

20. Burnie Blakeley, Harry Harris, and Rhys Lewis, *Messaging and Queuing Using the MQI* (Columbus, OH: McGraw-Hill, 1995).
21. Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Computing Surveys*, June 2003.

both send and receive messages. The server might actually consist of multiple servers arranged in some way. The network of servers is usually referred to as *the server*, when you're not interested in the server topology.

OO languages use their own lexicon with events. The .NET Framework describes events like this: *An event is a message sent by an object to signal the occurrence of an action.*[22]

There are two problems here. First, there is the confusion between events and notifications. The definition applies to notifications, not events. This confusion is common. People often use the word *event* when they are really talking about a notification, probably because what the sender calls a notification can be considered an event by the receiver.

The second problem is the word *action*, which is a bit misleading. An event might be triggered simply by the passage of time, which is hardly an action. In any case, the definition implies that .NET Framework notifications are messages, meaning they carry data of some sort. The .NET Framework documentation describes events as being *raised* or *sent* by the *event sender* and received by the *event target*, also called the *event consumer*. The notification payload is called the *event data*. In order to receive notifications, the event must be *wired* from the sender to the target. The method invoked in the consumer during the course of event firing is called an *event handler*.

In JavaBeans lingo, events are defined differently: *Events are objects sent from event sources to event listeners*. JavaBeans events are essentially the payloads of notifications. During registration, listeners are *added* to event sources. Listeners are said to *handle* events by providing methods that are invoked when the event source fires the event.

In Delphi Object Pascal, the definition is a bit more abstract: *An event is a mechanism that links an occurrence to some code.*[23] Here again, there is confusion. The definition alludes to both events and notifications.

In C++, there is no definition for the word *event*, as C++ predates the recognition of events as essential interaction mechanisms. C++ programmers today are likely to use the word *callback* when talking about events. A callback is the method called in a receiver during event notification.

CORBA technology uses yet another lexicon. In the CORBA event model, event sources and targets are known as event *suppliers* and *consumers*, respectively. Events travel over *channels*, which represent the abstract path from supplier to consumer. CORBA uses services to deliver events. The Event Service was introduced first and is used in systems where event filtering and customizable QoS are not necessary. The CORBA Notification Service is an extension of the Event Service, providing filtering and QoS.

Event Subscription

Event subscription is the process of linking an event publisher to an event subscriber. When a subscriber subscribes to an event, it essentially signs up to receive future notifications from the publisher. Event publishers are usually capable of detecting several types of events and sending different types of notifications. During the subscription process, the subscriber must identify the type of event it is interested in. The publisher stores the subscriber's subscription. Any time an event is detected, the publisher sends a notification to the subscriber of the event.

A publisher exposes the list of events it can detect. It exposes the list by *publishing* the events, which usually means that the events are made visible through interfaces, properties, or methods of some type. Publishers must be prepared to deal with the situation in which an event has no subscribers. Publishers must therefore possess a certain amount of intelligence to check for the existence of subscribers before trying to send notifications.

22. Events and Delegates, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconeventsdelegates.asp, 2004.
23. "What are events?," online help documentation for Borland Delphi 5, 1997.

A subscriber can usually subscribe to multiple types of events from the same publisher. Conversely, an event can have multiple subscribers. When the event is detected, the publisher notifies each of the subscribers. The order in which subscribers are notified depends on a variety of factors. I'll spend more time on events with multiple subscribers in Chapter 3.

In order to receive notifications, a potential subscriber must apply to the publisher for a subscription. In the simplest of systems, subscribers and publishers are connected directly. Subscribers subscribe directly by sending a subscription request to the publisher. The request must identify the subscriber, and the request can also contain other information, such as which types of events the subscriber is interested in, the subscriber's priority, and so on.

The Subscription Process

The subscription process includes everything that happens in order for a subscriber to become subscribed to one or more event notifications. The subscription process occurs at run time, never at compile time. If a subscription were made at compile time, you wouldn't call it a subscription and you wouldn't consider the interaction to be event-based. The subscription process can occur several ways, based on the arrangement of the publisher, the subscriber, and ancillary objects.

In the simplest arrangement, called the *direct-delivery model*, the subscriber submits subscriptions directly to the publisher. The publisher fires events directly to the subscriber, as shown in Figure 2-3.

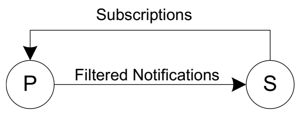

Figure 2-3. *Subscriptions in a direct-delivery system*

In larger systems, publishers might need to handle large numbers of subscribers or provide notification features that are computationally intensive. In such systems, a middleware system is often allocated to offload the event source from the notification overhead. Subscribers don't interact directly with the event publisher; they interact only with the middleware. Figure 2-4 shows the arrangement, called the *indirect-delivery model*, in which P represents a publisher, M represents a middleware system, and S represents a subscriber.

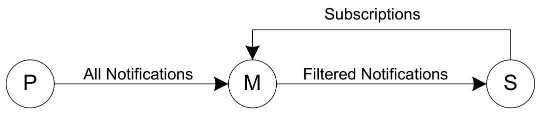

Figure 2-4. *Subscriptions in an indirect-delivery system*

With indirect-delivery systems, the subscriber sends subscription requests to the middleware system, which then assumes the responsibility of filtering and routing notifications to the subscriber. The middleware might receive notifications for all the events published by P, but only those that satisfy subscription criteria are sent to subscribers.

When it is important for subscribers to not be coupled to publishers or middleware, a different subscription process can be used: Instead of the subscriber requesting subscriptions itself, a separate *binder* agent makes the request. You can use binders in both direct- and indirect-delivery systems, as shown in Figure 2-5 and Figure 2-6.

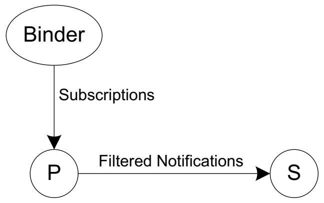

Figure 2-5. *Using a binder with the direct-delivery model*

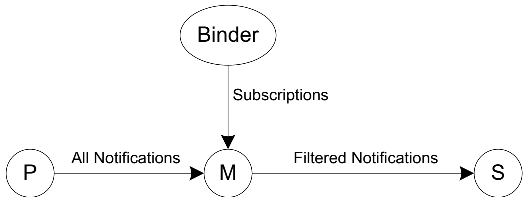

Figure 2-6. *Using a binder with the indirect-delivery model*

The binder takes its name from the fact that it provides the publisher with a *binding* to the subscriber for each notification identified in a subscription. The binder is coupled to both the publisher and the subscriber, while the subscriber is often completely decoupled from both the binder and the publisher and can be developed and tested on its own.

Subscription Models

The subscription model is based on how the subscriber identifies the events of interest. There are many ways to categorize subscription models. For example, JEDI[24] identifies four categories, called *subscription approaches*: channel subscriptions, subject subscriptions, content subscriptions, and event combination subscriptions. Each identifies the events of interest to the subscriber in a different way.

Many systems allow subscribers to be organized into *groups*.[25] The group is an entity that subscribes to certain events. Subscribers need only to specify which group or groups they are members of. The notion is similar to people signing up to newsgroups. Each newsgroup implicitly defines certain types of messages, so people determine the types of messages they get by choosing a newsgroup rather than defining specific message types.

The subscription models I use in this book appear in the hierarchy shown in Figure 2-7.

24. Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta, “The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS,” *IEEE Transactions on Software Engineering*, September 2001.

25. Daniel J. Barrett, Lori A. Clarke, Peri L. Tarr, and Alexander E. Wise, “A Framework for Event-Based Software Integration,” *ACM Transactions on Software Engineering and Methodology*, October 1996.

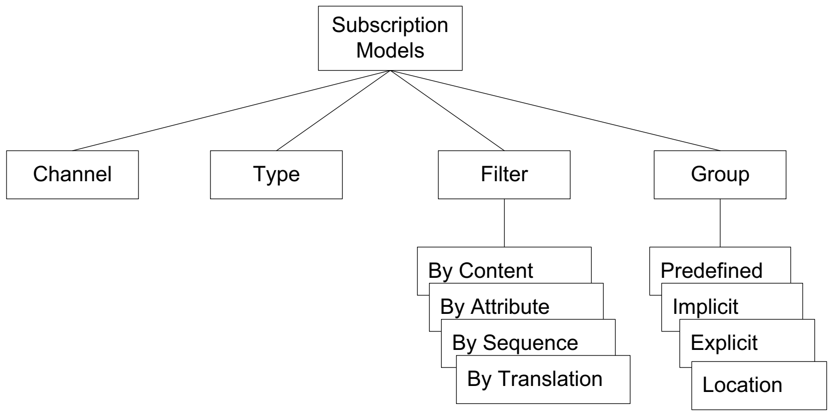

Figure 2-7. *The hierarchy of subscription models*

The four basic subscription models are *channel*, *type*, *filter*, and *group*. A *channel* denotes the physical or abstract construct through which notifications are sent to subscribers. A channel might be an outgoing procedure call, a network connection, a UML component port, or something else. Subscriptions by *type* are commonly used in direct-delivery EBSs developed with a component model (such as JavaBeans, .NET, or Delphi Object Pascal) that supports events or notifications as OO types. Filtered subscriptions use a *filter* to accept or reject notifications based on various criteria. A *group* represents a collection of subscribers that share the same subscription. Group subscriptions are useful when many subscribers are interested in getting the same notifications. The group would be associated to the proper channel or channels with the appropriate filters.

Channels

You can think of notification channels like television channels: Once you select a channel, you get all the programs broadcast on that channel. Event publishers can use any number of channels. If they use just one, then all notifications are sent on that one channel. More commonly, publishers define multiple channels and use an internal mapping algorithm to associate notifications to channels. Some publishers define a different channel for each type of possible notification.

A channel denotes the physical or abstract construct through which event notifications leave the publisher. In direct-delivery models, it is the responsibility of the event publisher to establish a mapping between events and channels. In indirect-delivery models, the notification middleware might be responsible for establishing the mapping. A channel can provide notifications for a single event or a group of events, as shown in Figure 2-8, Figure 2-9, and Figure 2-10.

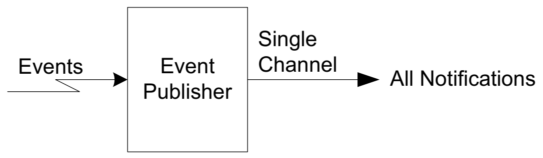

Figure 2-8. *A single channel carrying all notifications*

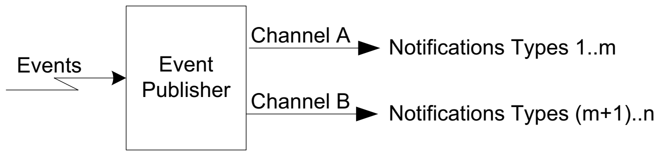

Figure 2-9. *Channels carrying multiple types of notifications*

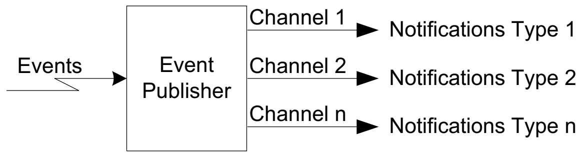

Figure 2-10. *Channels carrying one type of notification*

The event-to-channel mapping, also called *channelization*, can also be related to notification content. The need for channelization arises often when notifications contain textual information. For example, a news service might expose different channels for international news, local news, weather news, and so on, as shown in Figure 2-11.

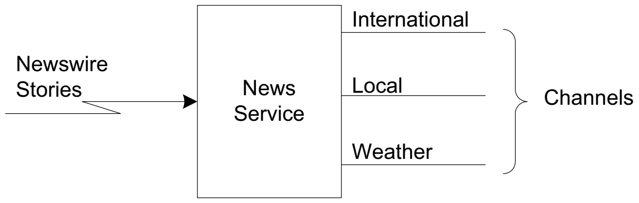

Figure 2-11. *Channels associated with content types*

The news service would need to extract information from incoming newswire stories and determine which channel to assign the story to. There can be a substantial amount of processing involved in determining which channel a notification belongs to. Large systems often use a separate system to handle the job, splitting the job of event detection and channelization, as shown Figure 2-12.

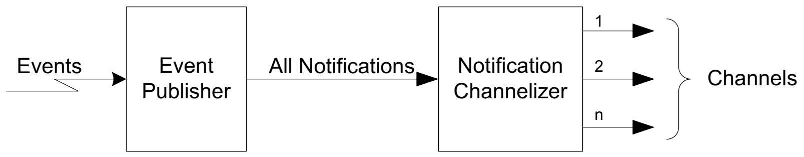

Figure 2-12. *Using a separate system to channelize notifications*

The event publisher detects events, while the channelizer uses an internal algorithm to route notifications to the proper channel.

Types

When talking about subscription models based on *types*, you need to be careful about what you mean with the word *type*, because the word is heavily overloaded in the software context. What is the *type* of an event?

In the JavaBeans event model, the event type refers to the object passed to an event listener's handler method. The event type is really the *payload type*. Several type-based notification systems organize event types this way, including the Cambridge Event Architecture[26] and Hermes.[27] In the .NET Framework event model, events are properties exposed by event publishers. Each event property represents a different event type. For example, a Button component might expose the three properties Clicked, Moved, and Resized. Delphi Object Pascal event types are similar. CORBA event types refer to the interfaces through which event producers call consumers.

Conceptually, a certain amount of overlap exists between channel-based and type-based subscriptions. Many systems associate channels with types, so that each channel relates to notifications of a given type. However, a channel can also carry many types of notifications, so type-based subscriptions are generally more selective and deterministic than channel-based ones.

Filters

Filter-based subscriptions are necessary when the subscriber wants only a subset of the notifications available. The subscriber specifies a filtering expression, and the publisher applies the filter to all the notifications generated. Those that aren't rejected by the filter are sent to the subscriber. When the filtering effort is relatively light, the publisher can often handle both the filtering and notification jobs, as shown in Figure 2-13.

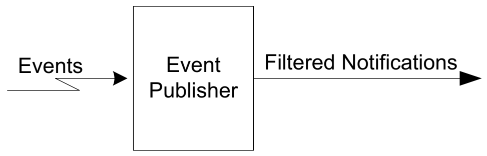

Figure 2-13. *Filtered notifications*

You can also use filters in conjunction with channels. In this case, the subscription identifies both the filtering expression and the channel identifying the notifications to filter. See Figure 2-14.

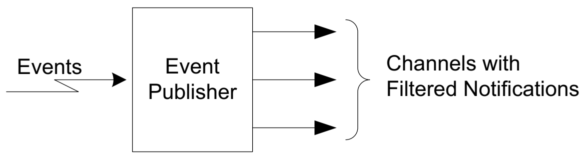

Figure 2-14. *Filtered notifications sent on diffferent channels*

26. Jean Bacon, Ken Mood, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, and Mark Spiteri, "Generic Support for Distributed Applications," *IEEE Computer*, March 2000.
27. Peter R. Pietzuch and Jean Bacon, "Hermes: A Distributed Event-Based Middleware Architecture" (proceedings of the International Conference on Distributed Computing Systems, Vienna, Austria, July 2002.

Filtering can be quite expensive computationally. In EBSs that support large numbers of subscribers or complex filtering, a dedicated system can perform filtering, as shown in Figure 2-15.

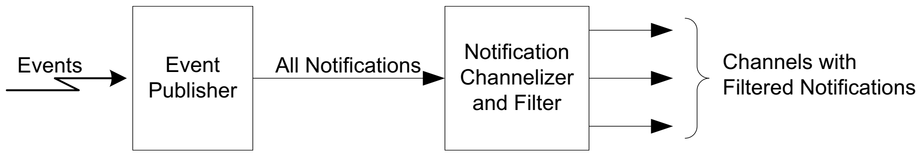
Events
Event Publisher
All Notifications
Notification Channelizer and Filter
Channels with Filtered Notifications

Figure 2-15. *Using a dedicated system to channelize and filter notifications*

To reduce the loading of the channelizing/filtering service, you can use a separate system to split notifications into channels, as shown in Figure 2-16.

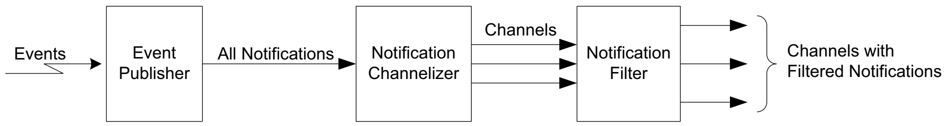

Figure 2-16. *Using separate systems to channelize and filter notifications*

Some systems use a special *subscription language* to specify filtering. Regardless of how the filtering criterion is specified, the effect of filtered subscriptions is to reduce the volume of notification traffic toward the subscriber. A notification can be associated with multiple filters, each specified by different subscribers, so notifications might have to be processed multiple times and using different filters in order to satisfy all subscribers.

Content Filtering

Content-based subscriptions apply to systems in which notifications always carry a payload representing the content. The subscription somehow describes what content is of interest. The notification system then applies the filter to the content of all incoming notifications. Only notifications that aren't rejected by the filter are sent on to subscribers. Using the news client example again, a content-based subscription model could allow the client to register only for news stories that contain a certain sequence of words, a regular expression, or a set of words.

A significant problem with content filtering is its computational cost: It is usually necessary to parse much (or even all) of a notification's payload to filter it properly and determine whether it passes the filtration criterion. Content filtering is feasible in systems with a small number of subscribers, but it can require substantial hardware resources in larger systems, such as Internet-scale ones. Gryphon,[28] JEDI,[29] and Siena[30] are examples of large-scale, content-based notification systems.

As a refinement of content-based filtering, Kulik[31] describes a hybrid content-based filtering model that relies on IP multicasting managed at the router level. Special systems called *match-structure*

28. Rob Strom, Guruduth Banavar, Tushar Chandra, Marc Kaplan, Kevan Miller, Bodhi Mukherjee, Daniel Sturman, and Michael Ward, "Gryphon: An Information Flow Based Approach to Message Brokering" (technical report, Hawthorne, NY, IBM T.J. Watson Research Center, 1998).
29. Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta, "The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS," *IEEE Transactions on Software Engineering*, September 2001.
30. Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service," *ACM Transactions on Computer Systems*, August 2001.

processors sit between publishers and multicast routers. The routers act as a distributed notification system. Processors read incoming messages and generate a header known as a *match-structure* that lets the routers determine the list of subscribers.

Attribute Filtering

In many systems, it is possible to classify events using a set of attributes. These attributes can be considered part of an n-dimensional *attribute space*. An attribute-based subscription identifies notifications of interest by specifying constraints in the attribute space. Attribute-based filtering also goes by different names in the literature, such as *topic-based* or *subject-based* filtering.

There are two fundamental types of attribute filters: those that work with preclassified events and those that must classify events on the fly, according to some criteria. The first type, also known as *source-side filtering*, requires the event source to add attributes to the event content. To simplify processing at the subscriber end, attributes are usually put in a special section of the notification payload, such as a header. The header might be expressed using Extensible Markup Language (XML) or a series of name-value pairs.

The second type of attribute filter puts the attribute extraction burden on the notification service, based on the event's content, history, traffic patterns, or other. If the attributes are based on keywords found in the content, the notification service must parse the content of every notification, which is computationally expensive. An even worse situation occurs when the attributes are based on event history or event patterns. Consider a stock exchange notification service that sends out real-time stock transactions. If a subscriber wants to be informed when the volume of a certain stock exceeds a certain value, an attribute-based subscription would be necessary. But to support the subscription, the notification service might need to parse every stock transaction, identifying the stock type and the number of shares involved. The service would also need to maintain internal state information to keep track of the total volume of shares handled of a given stock. Only when the volume hit the threshold set by the subscriber would a notification be sent. Clearly, such a system would be inefficient, expensive, and difficult to scale.

Attribute-filtering performance can be very good when the attributes are clearly defined in a content header. Such attribute-based filtering can have good enough performance to be used in embedded systems.[32] As an example of attribute filtering, consider a news server that receives stories classified by headline, date, author name, country, and name of the originating newswire service. The system's attribute space would have the dimensions {headline, date, author, country, wireService}. When specifying a filter, the subscription would use an expression that might look something like this:

```
(date: today) and (wireService: Reuters) and (location: Congo)
```

Attribute-based expressions are conceptually equivalent in many ways to the WHERE clause in a SQL query. Jin and Strom[33] describe a notification system that uses a relational database model to support SQL-like attribute-based filtering.

31. Joanna Kulik, "Fast and Flexible Forwarding for Internet Subscription Systems" (proceedings of the Second International Workshop on Distributed Event-Based Systems, San Diego, CA, June 2003).
32. Carlos Mitidieri and Jörg Kaiser, "Attribute-Based Filtering for Embedded Systems," (proceedings of the Second International Workshop on Distributed Event-Based Systems, San Diego, CA, June 2003).
33. Yuhui Jin and Rob Strom, "Relational Subscription Middleware for Internet-Scale Publish-Subscribe" (proceedings of the Second International Workshop on Distributed Event-Based Systems, San Diego, CA, June 2003).

Sequence Filtering

Subscribers might only be interested in a certain sequence of events. Subscriptions for sequence filters must identify the events in a sequence and their temporal constraints. Each event in a sequence can be further qualified by attribute, channel, or content restrictions. When events occur that fit the sequence filter, a notification is sent to the subscriber. Event sequences are also known as *composite events* in some systems, such as the Cambridge Event Architecture.[34] For example, a weather news subscriber might issue notifications of type `freewayTrafficAlert`, `precipitation`, and `freewayCondition`. A subscriber might be interested in getting `freewayCondition` notifications for freeway 405 in Southern California, but only if the freeway has traffic problems and it's raining. The subscriber would need to subscribe to `freewayCondition` notifications, but only those regarding freeway 405. To get the notifications only in the situations of interest, a sequence filter might be defined with an expression like this:

```
(freewayTrafficAlert.route: 405) and
(freewayTrafficAlert.date: today) and
(precipitation.location: california.southern) and
(precipitation.time.hour > now.hour - 6) and
(precipitation.amount > 0)
```

You could also use a sequence filter to reduce often-occurring notifications by specifying a cutoff frequency. For example, even if a notification occurs at the rate of three per second, a subscriber might only be interested in getting them once a minute. The subscription would use a filter specifying the maximum frequency of 1/60 Hz for the given notification.

Sequence filtering generally requires the management of internal state for each subscriber, which can result in a substantial amount of overhead for the filtering service. The filter state might be required to track the notification history for a given subscriber, such as the number and type of notifications already sent in a designated period of time.

Translation Filtering

Translation filters, also known as *message transforming functions*,[35] allow the following kinds of translations:

- *Notification content*: The filter might convert the content from one language to another or change the format of any dates found into a specific format. This type of filtering is obviously expensive, because it generally requires the entire payload to be processed.
- *Notification type*: This kind of filtering can be used to exclude specific types of notifications. Say a channel carries notifications related to events of type n1, n2, and n3. You could use a translation filter to block notification types that weren't of interest. You can also use notification type filtering to convert types. For example, a subscriber might wish for notifications of type n1, n2, and n3 to be converted to type n4, perhaps because the subscriber considers the three types to be equivalent.
- *Notification sequence*: This type of filtering is also known as *composite event detection*. The idea is to send a notification when a certain sequence of events has occurred. For example, a subscriber might want to be sent a notification n1 when events e1, e2, and e3 occur in a row and within five minutes of each other. In order to keep track of past occurrences of events, a translation filter needs to be stateful and maintain a separate state for each subscriber, which obviously can be computationally expensive in the presence of many subscribers.

34. Jean Bacon, Ken Mood, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, and Mark Spiteri, "Generic Support for Distributed Applications," *IEEE Computer*, March 2000.
35. Daniel J. Barrett, Lori A. Clarke, Peri L. Tarr, and Alexander E. Wise, "A Framework for Event-Based Software Integration," *ACM Transactions on Software Engineering and Methodology*, October 1996.

Groups

When subscribers sign up for the same events and use the same filters, it can be advantageous to use grouping to simplify the subscription and notification-delivery process. Groups act like virtual subscribers and can have an arbitrary number of members, including none. Figure 2-17 shows three members A, B, and C, in a group.

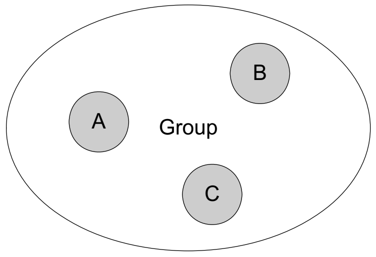

Figure 2-17. *Subscribers organized into a group*

A subscriber belonging to a group receives all the notifications the group is subscribed to. A subscriber might belong to multiple groups. Who determines which group or groups a subscriber belongs to? It depends on the system, but three general cases are possible:

- *The publisher*: This might determine the subscriber group based on something it knows about the subscriber, such the subscriber's name, role, age, or country.
- *The notification infrastructure*: This might determine the group based on the subscriber's connection speed, the node address, and so on. An important case is based on the subscriber's physical location. If the notification infrastructure defines *service areas* for different geographic areas or network nodes, the system might use this information to determine which notifications to send a subscriber.
- *The subscriber*: This might determine the group by specifying the group explicitly in the subscription, or indirectly by signing up for the same events as other subscribers.

A number of notification systems rely on groups, such as ISIS.[36] In general, there are no constraints on which events a group subscribes to, so groups may have an overlap in the notifications they specify, as shown in Figure 2-18.

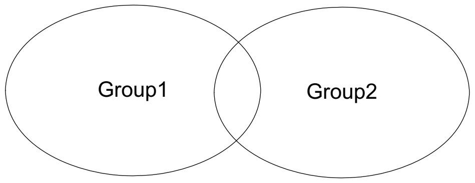

Figure 2-18. *Overlapping groups*

36. Kenneth P. Birman, "The Process Group Approach to Reliable Distributed Computing," *Communications of the ACM*, December 1993.

Members who belong to overlapping groups could receive duplicates of certain notifications, depending on the implementation. Groups can also contain other groups, recursively to any depth, as shown in Figure 2-19.

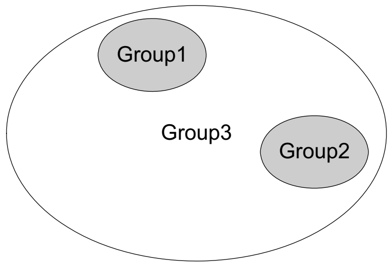

Figure 2-19. *Groups of groups*

The contained groups can conceivably have overlapping subscriptions, which could cause members of Group3 to receive duplication notifications.

Predefined Groups

Predefined groups are those that are defined by an event publisher or notification service. For example, a notification service for diagnostics events of a banking network might predefine three groups: administrator, supervisor, and technician. Each group would come with a built-in subscription for certain types of notifications. Members joining such groups can't change the group's subscription, but members wanting additional notifications beyond those provided by the group might be able to add their own private subscriptions.

Implicit Groups

Implicit groups are those that are set up automatically when two or more subscribers request the same subscription. The event publisher or notification service handling the subscription can use an optimized delivery mechanism to notify members of the group. Subscribers are generally not aware that they belong to an implicit group.

Explicit Groups

Explicit groups must be created at run time through a system-dependent operation. You can associate a group with any number of subscriptions. Once you create an explicit group, subscribers can join it as members. A group acts like a virtual middleware system, with its own subscriptions and list of subscribers. You can use such groups to classify subscribers in any number of ways, such as by role (e.g., administrator, supervisor, guest), creditworthiness (e.g., excellent, good, bad), and so on. A system using explicit groups must identify a policy for managing groups. Most systems identify a special administrator to set up and administer groups, barring normal subscribers from doing so.

Location Groups

In some systems, subscribers are connected to a network of middleware systems, as shown in Figure 2-20.

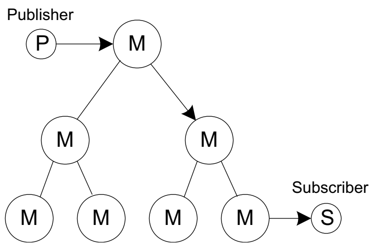

Figure 2-20. *A subscriber connected to a network of middleware systems*

The middleware systems are often situated in different geographic areas, and the middleware to which a subscriber is connected identifies the location of the subscriber. You can then associate each middleware with a predefined group, whose subscriptions determine the types of notifications sent to a subscriber. Such *location groups* are important, especially if subscribers are mobile. For example, a system could use location groups to distribute traffic information to wireless receivers in cars. Cars would only receive notifications pertinent to their location.

Subscription Policies

Subscriptions have a life cycle: They can be created, administered, and canceled. A subscription policy identifies who has the right to change subscriptions and how subscriptions behave over a given period of time. For example, a policy might issue subscriptions that are valid per a specific lease period, after which they expire. Such subscriptions could be useful when sending messages to users by e-mail. If a person never responds, the system could assume the person is not interested in the messages or has closed the e-mail account. In such cases, the person's subscription would expire at some point. The expiration policy might be based on criteria other than time, such as the total number of messages sent to a subscriber or the number of responses the subscriber sent. Other aspects defined in a policy are related to the following:

- Can subscriptions be changed?
- Can subscriptions be canceled?
- How many different subscriptions can a subscriber have?

When a subscription is changed or canceled, a certain amount of latency can exist between the time the change request is sent and the time the event publisher receives and processes it. During this time, which is often variable and related to system load, the publisher might send notifications that the subscriber is no longer interested in. The subscription policy should address this issue as well.

Summary

In this chapter, I provided a quick history and overview of events and notifications. Many people who develop "event-based" programs are unaware of the extensive amount of work and the number of people involved in making events what they are today. Contrary to popular belief, events were not invented by Microsoft for Visual Basic back in the early 1990s. However, Visual Basic is probably the first product that opened up the world of event-based programming to the masses. The adoption of the event-based model in Visual Basic is what made that product so much easier to work with, compared to other products and technologies of the day.