Java event delivery techniques

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

What is this tutorial about?

The Java 2 platform enables programmers to think about systems in terms of events, rather than a traditional call-return architectural style. Within the platform itself, there is direct support for several event notification patterns. One way to think about events is through the granularity and ownership of an event. Some event patterns succeed in presenting events at the granularity of a state change in an object instance. Others are more fine-grained and address changes in properties. Still other event techniques must be used to present events that are not necessarily associated with an object instance. Once mastered, event-programming techniques can be applied to user interface programming, service-oriented architectures, and enterprise application integration.

In this tutorial, we'll walk through a variety of event patterns and event delivery techniques that are tailored to specific event granularities. We'll start with relatively simple examples based on direct Java 2 platform classes and APIs, then work through the construction of a distributed event delivery service. By the end of the tutorial, you'll have learned first-hand about the complexities of building a distributed event environment.

Should I take this tutorial?

This tutorial is intended to guide you through the complexities of event usage on the Java 2 platform. To that end, we'll focus on building event mechanisms from scratch, rather than applying a more user interface-centric approach.

We'll work through various event patterns that deliver different granularities of events. Upon reaching a convenient granularity for building distributed systems, we'll extend the model to a distributed environment. We'll wrap up with a completely different type of event service known as a topic-based event service.

By the time you've completed this tutorial, you will be comfortable with a variety of event patterns and implementations. If you're interested in event mechanisms that are specifically built for user interfaces, you'll likely learn quite a bit from the examples, although we will spend most of our time implementing non-visual, model-centric event mechanisms.

The technologies and techniques we'll use to complete the exercises are as follows:

* **JavaBeans**: The JavaBeans technology illustrates many event patterns, but familiarity with JavaBeans technology is not a requirement for taking the tutorial. We'll walk through the examples of the important aspects of JavaBeans with respect to events.

* **Remote Method Invocation (RMI)**: By the latter half of the tutorial we'll implement a distributed event service. You need some basic knowledge of RMI technology to keep up with the example in this section.

* **UML**: Class diagrams produced with TogetherJ are used to illustrate class relationships in each section of the tutorial; you should be able to interpret these diagrams.

* **Design patterns**: Several design patterns and idioms are identified in the context of the

tutorial, but you need not be an expert on the patterns to follow the examples.

See Resources on page 30 for a listing of tutorials, articles, and other references that expand upon the material presented in this tutorial.

Code samples and installation requirements

All of the examples in this tutorial were built with the Netbeans development environment. While Netbeans offers first-class support for the JavaBeans event mechanism, any development editor or environment is sufficient to read the code.

It isn't essential that you compile and run the examples, but doing so may assist your learning. Java 2 platform, Standard Edition is required to compile and run the examples. See Resources on page 30 to download the Java 2 platform.

The binaries and source code for the examples used in this tutorial are available as a downloadable jar file. See Resources on page 30 to download the file.

About the author

Paul Monday is an Architect at J.D. Edwards. He has six years of hands-on Java platform experience in a broad range of technologies including J2EE, Jini, and Jiro technology. After graduation from Winona State University in Winona, MN, Paul completed a Master's degree in Computer Science from Washington State University in Pullman, WA. His focus for the Master's degree was operating systems, specifically the Linux operating system. After receiving his degree, Paul worked for IBM on the SanFrancisco Project, Imation Corp. on a storage appliance initiative, and Retek Inc. on an enterprise application integration (EAI) project.

Paul has written two books, *SanFrancisco Component Framework: An Introduction*, co-authored with James Carey and Mary Dangler; and *The Jiro Technology Programmer's Guide and Federated Management Architecture*, co-authored with William Conner from Sun Microsystems. Both books were published with Addison-Wesley. He has also written a variety of articles for print and online technical resources.

In his spare time, Paul has worked with Eric Olson on *Stereo Beacon*, a distributed MP3 player that uses a centralized event service and distributed event implementations for communication. He has also built a mobile agent framework for the Jiro technology platform. Paul can be reached at *pmonday@hotmail.com*.

Section 2. The Observer pattern

Overview

We'll start the tutorial with a discussion of the Observer pattern, which is directly implemented with a single class in the Java platform. The intent of the Observer pattern was identified by the Gang of Four (see Resources on page 30) as follows:

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

There are essentially two players in the Observer pattern:

* **Observable**: An object instance of this class generates events, typically based on data within the class (we'll reserve the use of the term *properties* for our discussion of JavaBeans patterns).

* **Observer**: This is an interface that is implemented by classes that want to register themselves with **`Observable`** object instances. Classes that implement the **`Observer`** interface must implement a **`notify()`** method.

The Observer pattern in an event model

The Observer pattern illustrates several components of an event model:

* **Event source**: The object that originates events

* **Event listeners**: The set of objects that listen and, typically, react to events

* **Event delivery process**: The process that the event source uses to deliver an event to the listeners

* **Event data**: Data that accompanies an event notification, allowing a listener to act based on the context in which the event occurred

* **Event registration process**: The mechanism that a listener uses to register as an interested party to events that can occur in the system, or specific object source

Note that the above list references three class types

* Source
* Listener
* Data

and two processes:

* Delivery
* Registration

As we vary one or more elements of an overall event system, the other attributes of the event system will vary with it.

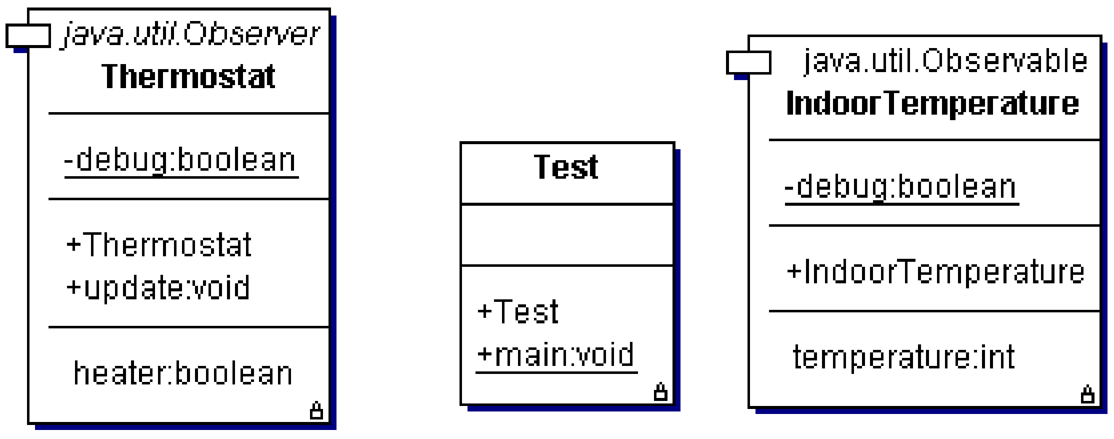
The Observer pattern in the Java 2 platform

As previously mentioned, the Observer pattern in the Java 2 platform is built with a single class, the **`Observable`**, and an interface, the **`Observer`**. An implementation of the **`Observer`** interface is notified of an event through a call to its **`update()`** method by the class that extends the **`Observable`** implementation. The Java 2 platform supplies a variety of methods for handling the relationship between observers and observables. We'll use the following subset of methods for our first example:

* **`setChanged()`**: This method is called by an **`Observable`** implementation when something has changed within the **`Observable`** (assuming this change is relevant to the observers).
* **`notifyObservers()`**: This method is called to notify observers of a change to the **`Observable`**. There are two implementations of this method: one that takes no arguments and one that allows the **`Observable`** to package a **`Data`** object to be forwarded to the observers. If the **`setChanged()`** method is not called, then the **`notifyObservers`** methods will not cause an event to be delivered to the observers.
* **`addObserver(Observer)`**: This method is called to register an object as an observer. Once registered, it will be notified of changes to the object state.

Observer example: IndoorTemperature

To illustrate the use of the Observer pattern in the Java 2 platform, we'll use a simple example, an application to monitor the temperature in an indoor environment. The classes are illustrated in the UML diagram below:



You should note the following about the classes and methods in the diagram:

* **`IndoorTemperature`** is a subclass of **`Observable`**, functioning as the event source.

* Within the **`setTemperature()`** method, the **`notifyObservers()`** method will be called after **`setChanged()`**. This notifies all observers of a temperature change.

* **`Thermostat`** implements the **`Observer`** interface and is registered with the **`Observable`** instance.

Not identified in the class diagram is a driver program that registers the thermostat with the indoor temperature.

The event source: IndoorTemperature

The code for the **`IndoorTemperature`** class is shown below. Within the constructor you will see that the temperature goes up and down between a maximum and minimum range, in one-second intervals. This is the case for all temperature implementations in the tutorial.

Next, look at the **`setTemperature`** method. Note that after the temperature is changed, the inherited **`setChanged`** method is called. This tells the **`Observable`** that the observers should be notified of an object state change.

Finally, the **`notifyObservers`** method is called; this method will *serially* notify all observers that are registered for notifications. It is important to note that the **`notify`** call blocks as it notifies each observer. As a result of this, an ill-behaved observer could consume the thread, not allowing others to be notified of the occurrence. Because the event-delivery mechanism is provided for us in this example it is difficult to modify the observer's behavior. Later in the tutorial we'll implement our own event-delivery mechanism. At that point we'll be able to address the observer's behavior.

```
public class IndoorTemperature extends java.util.Observable {
      private int temperature;
      private static final boolean debug = true;
      /** Creates new IndoorTemperature */
      public IndoorTemperature() {
          new Thread(new Runnable(){
              Monitor m = new Monitor();
              public void run(){
                  boolean increment=false;
                  int max=67;
                  int min=62;
                  int cur=67;
                  while(true){
                      if(increment) cur++;
                      else cur--;
                      setTemperature(cur);
                      if(cur==min) increment = true;
                      else if(cur==max) increment=false;
                      m.block(1000);
                  }
              }
          }
          ).start();
      }
      public int getTemperature() {
          return temperature;
      }
      public void setTemperature(int temperature) {
          if(this.temperature != temperature){
              if(debug) System.out.println(
                  "INFO::IndoorTemperature::setTemperature "
                  +temperature);
              this.temperature = temperature;
              setChanged();
          }
          notifyObservers();
      }
}
```

The event listener: Thermostat

The event listener, in this case a thermostat, will be notified when a temperature change occurs. Based on the temperature on the indoor temperature class instance, the thermostat will determine whether the heater should be turned on or off. The **update** method is called by the **Observable** class, **IndoorTemperature**, when a notification occurs. See the code listing below:

```
 public class Thermostat implements java.util.Observer {
        private boolean heater;
        private static final boolean debug = true;
        public Thermostat() {
        }
        public void update(java.util.Observable observable, java.lang.Object obj) {
            if(observable instanceof IndoorTemperature){
                IndoorTemperature it = (IndoorTemperature)observable;
                if(it.getTemperature() < 64) {
                    setHeater(true);
                } else {
                    setHeater(false);
                }
            }
        }
        public boolean isHeater() {
            return heater;
        }
        public void setHeater(boolean heater) {
            if(heater != this.heater){
                if(debug) System.out.println("INFO::Thermostat::setHeater"+heater);
                this.heater = heater;
            }
        }
 }
```

Registering the Observer

The main program that drives the example first creates instances of the **IndoorTemperature** class and the **Thermostat** class. After creation, the **Thermostat** object is registered with the **Temperature** class through the **addObserver** method. See the code listing below:

```
            // Create a new Observable
            IndoorTemperature it = new IndoorTemperature();
            // Create a new Observer
            Thermostat t = new Thermostat();
            // Register the Observer with the Observable
```

```
          it.addObserver(t);
```

As in all event systems, note that **`Thermostat`** is driven by the event mechanism. In this case, the system is still tightly coupled. The **`Thermostat`** must be explicitly registered with each event source. In later examples, you'll learn about ways to listen for events without knowing the exact event sources at compile time.

Running the example

Here you can see the results of an example run of the Observer test program. Note that the program will run forever unless you kill it using CTRL-C on Windows, or a similar tactic on another platform.

```
C:\java\tootomatic\events>java -classpath events.jar
                                 com.stereobeacon.events.observer.Test
INFO::IndoorTemperature::setTemperature 66
INFO::IndoorTemperature::setTemperature 65
INFO::IndoorTemperature::setTemperature 64
INFO::IndoorTemperature::setTemperature 63
INFO::Thermostat::setHeater true
INFO::IndoorTemperature::setTemperature 62
INFO::IndoorTemperature::setTemperature 63
INFO::IndoorTemperature::setTemperature 64
INFO::Thermostat::setHeater false
INFO::IndoorTemperature::setTemperature 65
INFO::IndoorTemperature::setTemperature 66
INFO::IndoorTemperature::setTemperature 67
INFO::IndoorTemperature::setTemperature 66
INFO::IndoorTemperature::setTemperature 65
INFO::IndoorTemperature::setTemperature 64
```

Summary

In this section we walked through an example of a primitive, yet effective event pattern, the Observer. The observable's delivery mechanism is *synchronous*: it notifies each observer in sequence using the **`update`** method, which is coded by an implementor of the **`Observer`** interface. The level of granularity for notifications is the object. If any part of the object changes, the listeners should be notified of this change.

One weakness of this event model is that the listeners must be aware of all object instances from which they want to receive notifications. This creates a tightly coupled system. A second weakness is the vulnerability of the event-delivery mechanism. The observer can consume the delivery thread. Finally, the listener cannot be registered for specific events. The listener is responsible for determining the change of state that caused the observable to fire a notification. This problem can be slightly alleviated by packaging an object with the notification mechanisms.

Despite these weaknesses, the Java 2 platform's Observer pattern implementation is a solid foundation for many small Java applications.

Section 3. JavaBeans patterns

Overview

Now that we've laid the groundwork for event mechanisms with the Observer pattern, we can quickly morph our example to illustrate different techniques for event systems. The two patterns we'll work with in this section use the JavaBeans patterns built into the Java 2 platform. These patterns are based on the publish-subscribe model. They are as follows:

* **Bound Property pattern**: This pattern creates a fine granularity for an event, a single property.

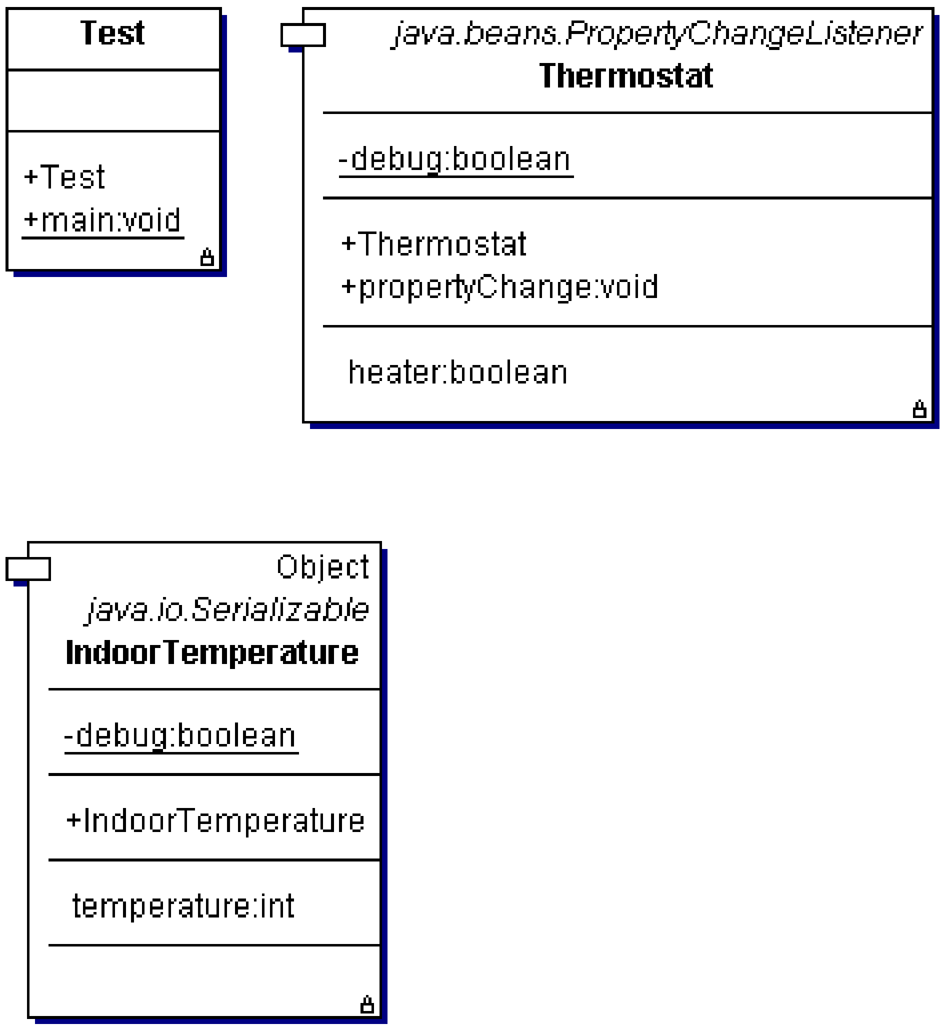
* **Multicast Event pattern**: This pattern decouples JavaBeans events from the granularity of the property. It also establishes a pattern for registering specific events. With the Multicast Event pattern the granularity of an event becomes the event itself, coupled with the object instance that fired the event.

As we go through these examples, you should keep in mind two things. The first is that the JavaBeans patterns are convenient for integrating with tools. The second is that we are looking at the event mechanisms themselves for ways to build event systems. The JavaBeans patterns show many attributes necessary for different types of event-based systems. We'll briefly discuss the many JavaBeans event patterns available at the end of this section.

The Bound Property pattern

Bound properties in JavaBeans allow a listener to register for notifications against a property of the JavaBean. For example, if the temperature on our indoor temperature gauge is a bound property, then you can register for change notifications with the JavaBeans component.

The UML diagram below offers an overview of the classes we'll use for this example:



In the panels that follow, we'll walk through the changes we need to make to modify our example to use the Bound Property pattern.

Adding support for bound properties

To support bound properties, you can use an instance of the **`PropertyChangeSupport`** class in the **`java.beans`** package. This keeps track of listeners based on a key that represents the property. It works much like a hashtable that is wrapped and ready for properties. We add the following declaration to the class scope. We add the constructor call to create an instance of the class within our own constructor (this is the only modification of the constructor for this exercise):

```
private PropertyChangeSupport propertySupport;
public IndoorTemperature() {
    propertySupport = new PropertyChangeSupport ( this );
    // thread code ...
}
```

Under this model we no longer inherit from **`Observable`**, so we add our own subscription methods, as shown here:

```
public void addPropertyChangeListener (
    PropertyChangeListener listener)
{
    propertySupport.addPropertyChangeListener (listener);
}
public void removePropertyChangeListener (
    PropertyChangeListener listener)
{
    propertySupport.removePropertyChangeListener (listener);
}
```

Notice from the above code that we had to change the listener interface to **`PropertyChangeListener`**. We'll come back to this in a moment. Also note that we're not using the support for keyed properties with the property change support class.

Altering the property change method

The next change we have to make to the **`IndoorTemperature`** class, the event source, is to alter the property change method to fire a property change event, as shown here:

```
public void setTemperature(int temperature) {
    int oldTemperature = this.temperature;
    this.temperature = temperature;
    if(debug) System.out.println(
            "INFO::IndoorTemperature::setTemperature "+temperature);
    propertySupport.firePropertyChange(
            "temperature",
            new Integer(oldTemperature),
            new Integer(temperature));
}
```

Notice the event data for a property change event. As shown above, the event data consists of the property being changed, the new value for the property, and the old value for the property.

Upgrading the registration process

Finally, the registration process in the main program has to undergo some change in order to work with the JavaBeans Bound Property pattern. We rewrite the registration method as shown below:

```
    IndoorTemperature it = new IndoorTemperature();
    Thermostat t = new Thermostat();
    it.addPropertyChangeListener(t);
```

Observer versus Bound Property pattern

As previously mentioned, as a result of our efforts, listeners now have to implement the **`PropertyChangeListener`** interface rather than the **`Observer`** interface. The logic is essentially the same, however, as illustrated here:

```
public void propertyChange(PropertyChangeEvent propertyChangeEvent) {
```

```
           if(propertyChangeEvent.getPropertyName().equals("temperature")){
               int temp =
((Integer)propertyChangeEvent.getNewValue()).intValue();
               if(temp < 64) {
                   setHeater(true);
               } else {
                   setHeater(false);
               }
           }
       }
```

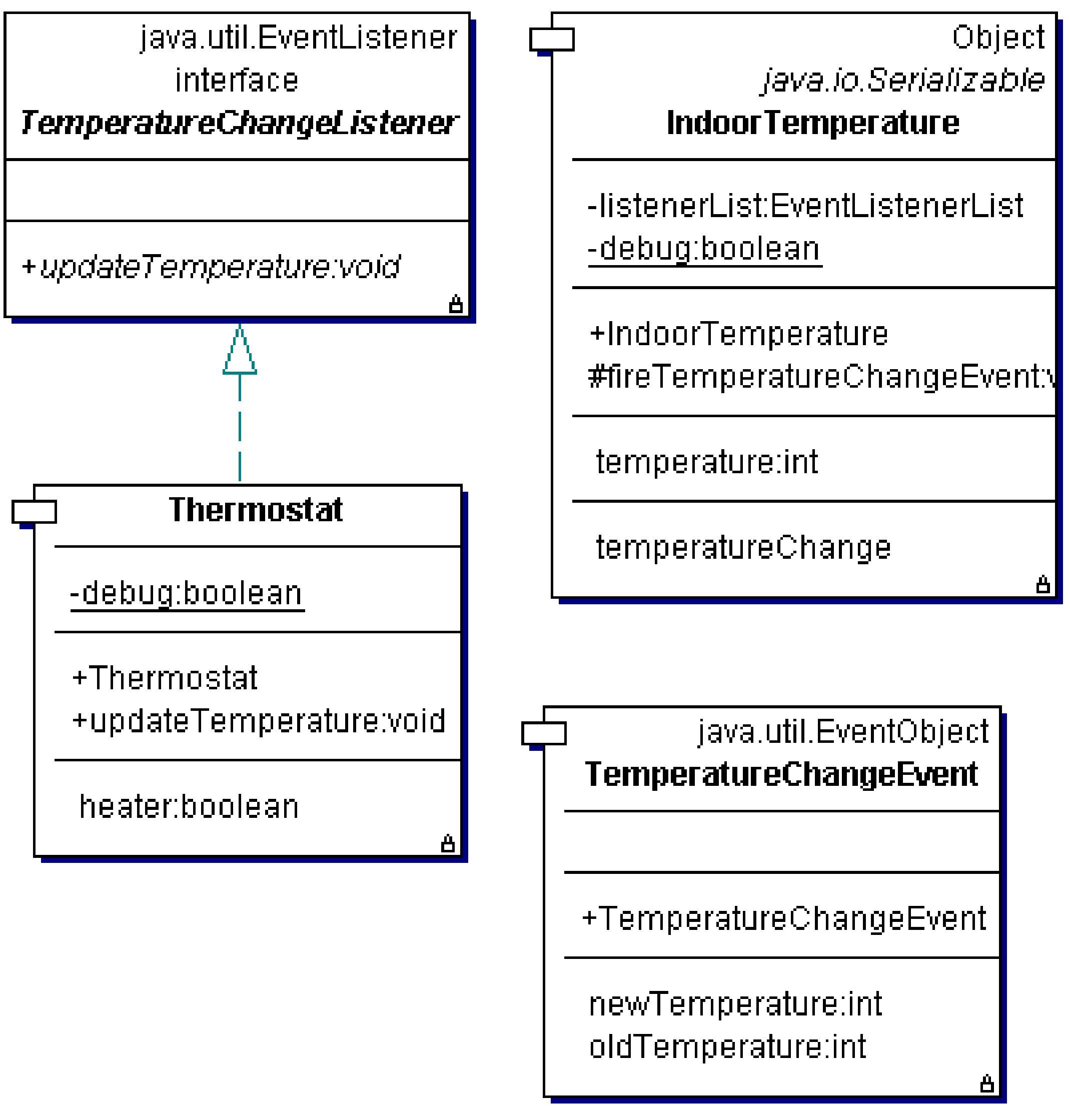
Running this program yields the same results as running the previous program. The source code for the Bound Property example is in **`com.stereobeacon.events.javabeans`** (see Resources on page 30). Initiate the test program by using the main class, **`com.stereobeacon.events.javabeans.Test`**.

The Multicast Event pattern

Bound properties are incredibly powerful for an IDE environment. They allow graphical environments to connect an event with an action. The downside to bound properties is that an event must be associated with a specific property. Consider what would happen if you wanted to surface internal exceptions as events, possibly by hooking up your own monitoring tools. You certainly wouldn't want to add properties for every exception that could occur!

The Multicast Events pattern lets us move toward a more generalized event model with the granularity of an event rather than the property of an object. Decoupling events from properties greatly extends the ways we can communicate occurrences in a system.

The UML class diagram below illustrates the classes we'll use for this example:



In the panels that follow, we'll go over the changes we need to make to introduce multicast events to our IndoorTemperature application.

Custom-built event data objects

The first major change is that from here on we'll declare our own event data objects. In the Observer pattern you were able to pass an instance of **`Object`** as data. In the Bound Property pattern you passed the old and new value. From this point forward we'll be passing custom-built event data objects. As you can see from the listing below, however, this isn't much of a change from working with the contents of a bound property:

```
public class TemperatureChangeEvent extends java.util.EventObject {
      private int newTemperature;
      private int oldTemperature;
      public TemperatureChangeEvent(Object source, int oldTemp, int newTemp){
          super(source);
          oldTemperature = oldTemp;
          newTemperature = newTemp;
      }
      public int getNewTemperature() {
          return newTemperature;
      }
      public void setNewTemperature(int newTemperature) {
          this.newTemperature = newTemperature;
      }
      public int getOldTemperature() {
          return oldTemperature;
      }
      public void setOldTemperature(int oldTemperature) {
          this.oldTemperature = oldTemperature;
      }
}
```

Implementing a custom interface

In addition to the event data change, the `Observer` will have to implement a custom interface to receive event notifications, as shown in the following code:

```
public interface TemperatureChangeListener extends java.util.EventListener {
        public void updateTemperature(TemperatureChangeEvent evt);
}
```

Changes to the event source

The remainder of the changes to support the JavaBeans Multicast Event pattern occur in the event source. We'll modify the event firing mechanism and the event registration mechanism.

We'll start with writing our own custom event firing mechanism. Writing our own firing mechanism is useful, because we can later adjust it to fire events asynchronously or implement a variety of patterns for acknowledging listener activities.

In the listing below, you can see a new, custom-built event firing mechanism:

```
public class IndoorTemperature extends Object implements java.io.Serializable {
        // some code removed for brevity
        private EventListenerList listenerList =  null;
        public void setTemperature(int temperature) {
            fireTemperatureChangeEvent(this.temperature, temperature);
            this.temperature = temperature;
        }
        protected synchronized void fireTemperatureChangeEvent(
            int oldTemp,
            int newTemp)
        {
            if(listenerList != null){
                TemperatureChangeEvent tce =
                    new TemperatureChangeEvent(this, oldTemp, newTemp);
                EventListener[] list =
                    listenerList.getListeners(TemperatureChangeListener.class);
                for(int i=0 ; i<list.length ; i++){
                    TemperatureChangeListener tcl =
                            (TemperatureChangeListener)list[i];
                    tcl.updateTemperature(tce);
                }
            }
        }
        public synchronized void addTemperatureChangeListener(
            TemperatureChangeListener listener)
        {
            if (listenerList == null ) {
                listenerList = new javax.swing.event.EventListenerList();
            }
            listenerList.add(TemperatureChangeListener.class, listener);
        }
        public synchronized void removeTemperatureChangeListener(
            TemperatureChangeListener listener)
        {
            listenerList.remove(TemperatureChangeListener.class, listener);
        }
  }
```

Registration methods

The class **`javax.swing.event.EventListener`** class is a powerful class for organizing event listeners that will be used for the remainder of the tutorial. It allows you to keep track of listeners by the **`Listener`** interface. So, if you have a single class that can allow you to register multiple event types, you can easily track the listeners.

The listener list is kept in the **`listenerList`** instance variable, which is declared as:

```
        private EventListenerList listenerList =  null;
```

We add the following methods to the **`IndoorTemperature`** class to register for temperature changes:

```
        public synchronized void addTemperatureChangeListener(
            TemperatureChangeListener listener)
        {
            if (listenerList == null ) {
                listenerList = new javax.swing.event.EventListenerList();
            }
            listenerList.add(TemperatureChangeListener.class, listener);
        }
        public synchronized void removeTemperatureChangeListener(
            TemperatureChangeListener listener)
        {
            listenerList.remove(TemperatureChangeListener.class, listener);
        }
```

With only minor changes to the main program and the observer class, **`Thermostat`**, our program is up and running again, this time using the Multicast Event model.

The source code for this example is in the **`com.stereobeacon.events.messages`** (see Resources on page 30). The test program can be run by using the main class, **`com.stereobeacon.events.messages.Test`**.

Additional JavaBeans event patterns

The examples used in this section are only two of the numerous JavaBeans event patterns.

Here are some of the notable variations:

* **Unicast Event pattern**: A simple variation of the Multicast Event pattern, but worthy of a brief mention. Basically, it establishes a one-to-one relationship between event publisher and the event listener (rather than a one-to-many relationship).

* **Constrained Property pattern**: Very similar to the Bound Property pattern, but the property is constrained to be within a specified range of values.

* **Vetoable Property pattern**: Allows event listeners to "vote" on whether a property can be changed to a new value. Essentially, each listener has the option to accept the property change or throw an exception. If an exception is thrown, the property's value cannot be altered and must be reset to the original value.

Summary

The JavaBeans patterns supply us with more places to learn about event publishing and subscription, as well as the mechanisms that must be implemented in event-based systems. We started with bound properties and moved to multicast events. Bound properties are associated with specific properties of a JavaBean, whereas multicast events need not be associated with a property (though in our example the association has been maintained). The JavaBeans event patterns are primarily targeted at IDEs and tools integration.

We also learned about a class in the **`javax.swing.event`** package that helps organize listeners. This will come in handy when it comes to working with classes that serve multiple event types and listeners.

Despite the wide variety of different event types supported by the JavaBeans package, there are still some flexibility issues that we can resolve as we move to a more dynamic, loosely coupled system. For example, listeners and event sources must register with each other explicitly. What this means is that we couldn't start having multiple event sources (such as an upstairs and a downstairs temperature) without registering with both of the temperature gauges explicitly. The event source is identified by the event type *and* the specific object instance firing the event. This isn't very flexible. In the next section we'll resolve this problem by decoupling the event-firing mechanism from the event source.

Section 4. Decoupling the event service

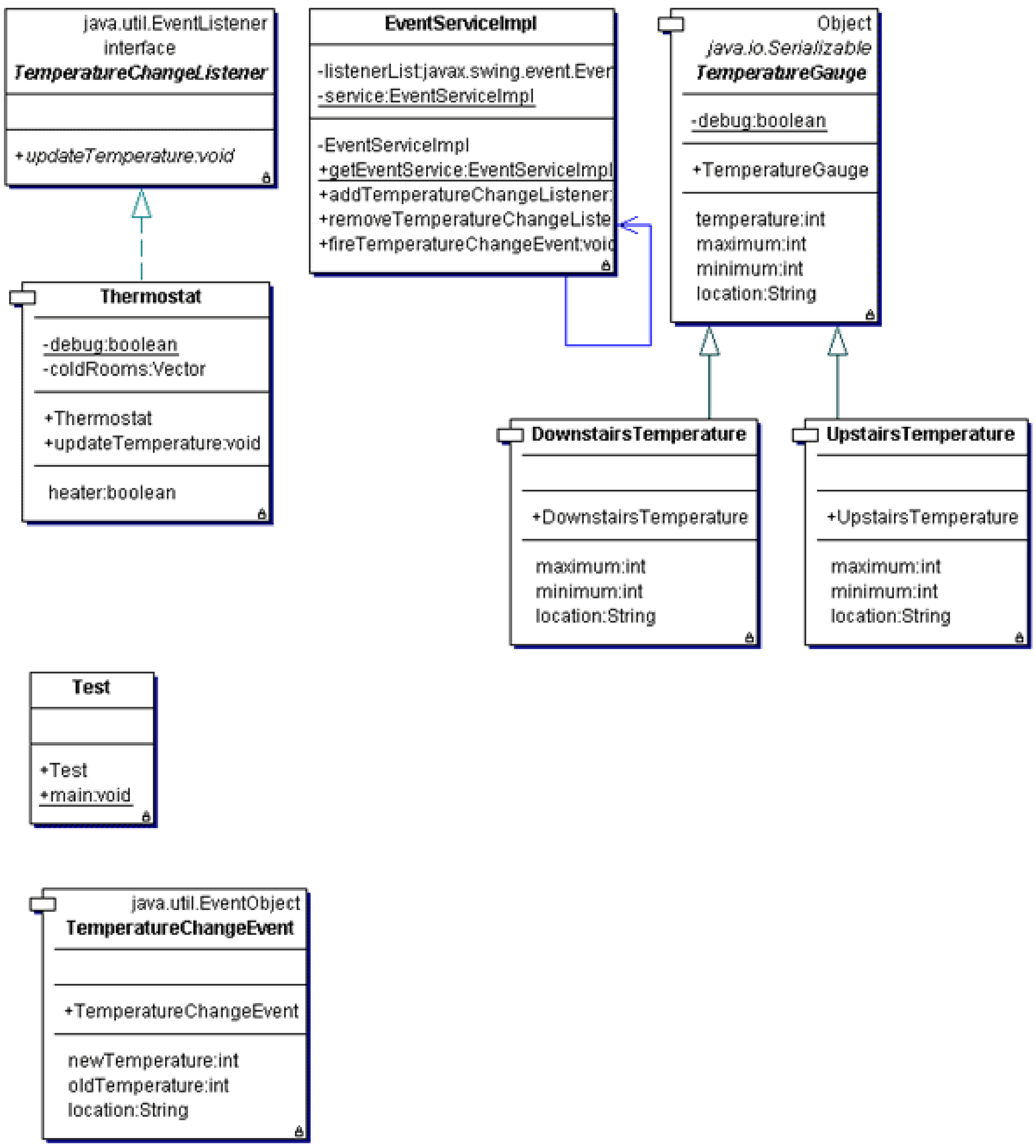
Overview

One of the downsides of the multicast event mechanism in JavaBeans is that event listeners must be aware of all of the event sources they are interested in, and register with each of them. A second weakness is that the behavior of event delivery is controlled by the individual event sources, which could become difficult to manage.

We can address these weaknesses by creating a stand-alone event service. The stand-alone service will contain the registration mechanism and the delivery mechanism, *plus* a public mechanism to submit an event for delivery. The major change in our current classes occurs in the event source. It will no longer track listeners; instead, upon having an event to fire, it will get a reference to the event service and call the service's firing mechanism.

In addition to creating a stand-alone event service, we'll add a second event source to our IndoorTemperature example. Through this exercise, you'll discover how simple it can be to add and remove event sources without affecting the event listener or the listener registration process. This has a dramatic effect in a distributed environment.

The UML class diagram below illustrates the classes we'll implement in this section:



Creating a stand-alone event service

For the most part the code for the stand-alone event service is the same as the multicast event source, but with the logic specific to handling the temperature property removed. The one major change is that we've presented the event service as a Singleton design pattern.

To implement the Singleton pattern, we change the constructor to a private constructor and place a static **`get()`** method on the event service to return the single instance. We do this to avoid a problem that can occur when an event source fires an event and the listener is registered to a separate event service. (On the other hand, if we planned ahead we could have separate event services for different event groupings.)

The second major alteration is to the **`fireTemperatureChangeEvent`**; we change it to be a public method on the event service rather than a protected (or potentially private) method, as it was in the JavaBeans implementation:

```
public class EventServiceImpl {
      private EventListenerList listenerList =  null;
      private static EventServiceImpl service = null;
      private EventServiceImpl() {
      }
      public static EventServiceImpl getEventService(){
          if(service==null){
              service = new EventServiceImpl();
          }
          return service;
      }
      public synchronized void addTemperatureChangeListener(
          TemperatureChangeListener listener)
      {
          if (listenerList == null ) {
              listenerList = new javax.swing.event.EventListenerList();
          }
          listenerList.add(TemperatureChangeListener.class, listener);
      }
      public synchronized void removeTemperatureChangeListener(
          TemperatureChangeListener listener)
      {
          listenerList.remove(TemperatureChangeListener.class, listener);
      }
      public void fireTemperatureChangeEvent(TemperatureChangeEvent event) {
          if (listenerList == null) return;
          Object[] listeners =
                  listenerList.getListeners(TemperatureChangeListener.class);
          for (int i = 0 ; i<listeners.length; i++) {
((TemperatureChangeListener)listeners[i]).updateTemperature(event);
          }
      }
}
```

Performance tuning the event service

Several factors could hurt the performance of our event service. Foremost is the fact that multiple event sources and listeners attempt to use the event service at the same time. If we oversynchronize the event service, we'll create a bottleneck; if we undersynchronize it we'll have problems with corruption. Another problem is that our listeners could easily take over the event service if the firing event is synchronized.

One way to sustain the throughput and loosen our dependency on the listeners' good behavior is to create a thread pool and spawn threads for each call to a listener. This technique creates a nice asynchronous model.

Finally, the interface may become very cluttered, depending on how many events we have flowing in our application. For a more compact solution, we may want to create more generic events with specific types filtered at the listener, rather than in the event service. Another technique for working around this problem is the topic-based event service, which we'll look at in more detail later in the tutorial.

Changes to the event sources

There are three major changes to the event sources in this section. The first is insignificant to the event model we're using. If you open up the source code that accompanies the tutorial, you'll notice that the **`IndoorTemperature`** class now inherits from **`TemperatureGauge`**. **`TemperatureGauge`** contains 95 percent of the temperature-management methods. Each temperature gauge subclass is responsible only for setting the boundaries of the maximum and minimum temperatures. For the purpose of this tutorial, we could now add temperature gauges very quickly with a few lines of code in the subclass.

The second change to the temperature gauge class is the removal of all listener-management methods and instance variables that are specific to listener management. This process is now handled by the stand-alone event service.

The final change is to the **`setTemperature`** method, as shown below. Now, instead of calling internal methods for delivering the event, we'll obtain a reference to the event service and call the service's firing mechanism:

```
public void setTemperature(int temperature) {
    if(debug) System.out.println(
            "INFO::"+getClass().getName()+
            "::setTemperature "+temperature);
    TemperatureChangeEvent tce =
            new TemperatureChangeEvent(
            this,
            getLocation(),
            this.temperature,
            temperature);
    this.temperature = temperature;
    EventServiceImpl.getEventService().fireTemperatureChangeEvent(tce);
}
```

Changes to the event listeners

We've made no changes to the event listeners themselves. There is, however, a small change to the main program. Instead of registering with the event source, we'll register with the event service. as shown here:

```
TemperatureGauge up = new UpstairsTemperature();
TemperatureGauge down = new DownstairsTemperature();
// Create a new Observer
Thermostat t = new Thermostat();
// Register the Observer with the Observable
EventServiceImpl.getEventService().addTemperatureChangeListener(t);
```

Note that we have registered the **`Observer`** only a single time and it will receive event notifications from both of the temperature gauges. The listener granularity is the event, *not* the event and the object instance.

The source code for this example is in the **`com.stereobeacon.events.standalone`** (see Resources on page 30). Run the test program by using the main class, **`com.stereobeacon.events.standalone.Test`**.

Summary

By converting to a stand-alone event service within our single process we've centralized the event registration mechanism and delivery processes. This makes it easier to modify the event processes, and it lightens the burden on class implementors that want to leverage events.

The change is especially convenient for listeners that are implemented without knowledge of how many or what objects will fire events. We no longer need to know all instances that will originate events.

There are downsides to centralized event mechanisms, however. They are not especially friendly to tool environments, and there are now multiple objects involved in the event-delivery process rather than the single observer and observable instances.

Section 5. Creating a remote event delivery service

Overview

Building distributed applications can be a difficult and daunting task, even with the Java 2 platform. It typically isn't getting the first system running that is difficult; it is making the system robust and failsafe that causes the problem.

In this section, we'll convert the stand-alone event service built in the last section into a stand-alone distributed event service. By the end of this section, we will have an event service that runs in its own JVM and can be reached by any event source or event listener by simply locating the event service in the network.

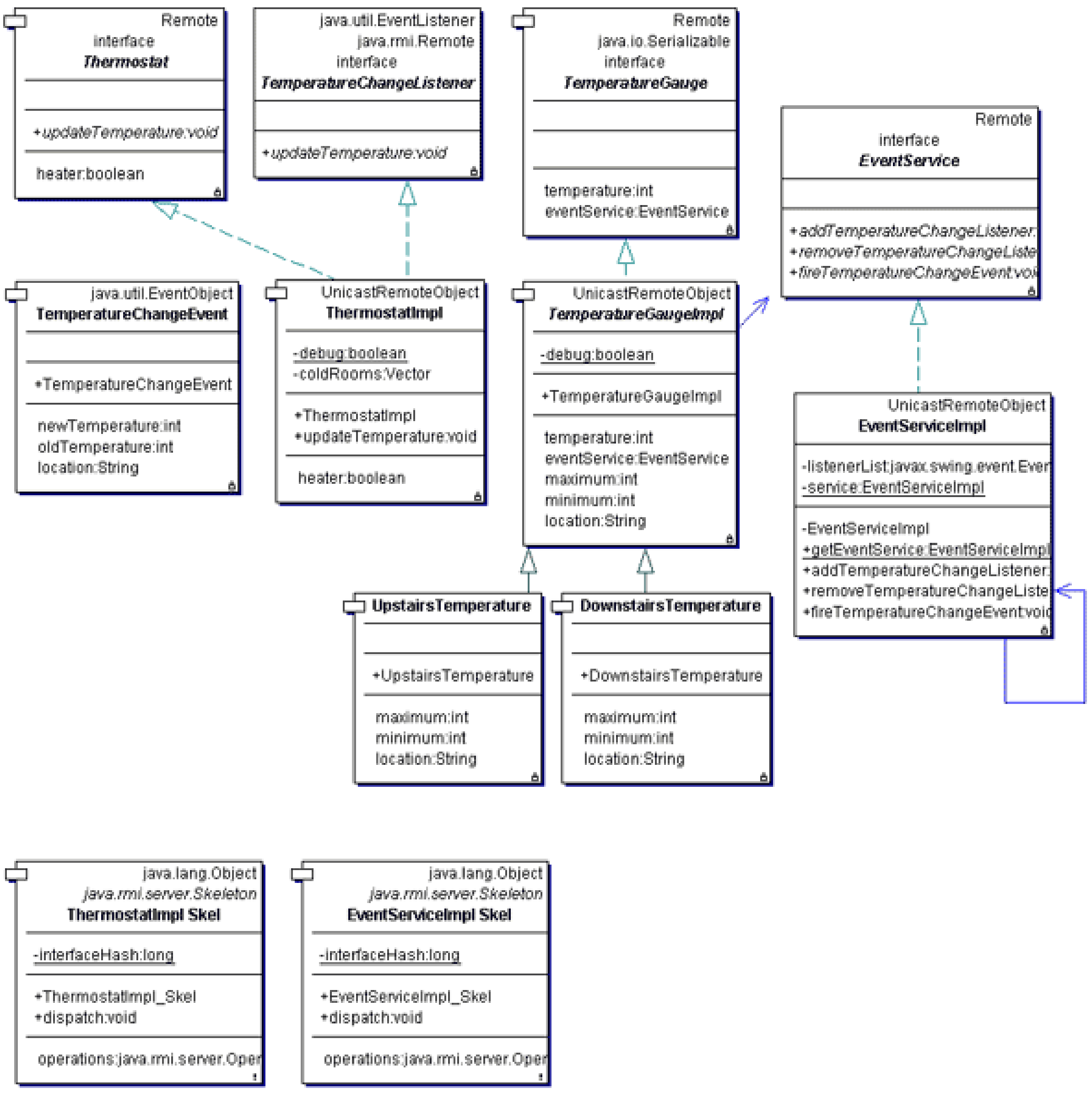
Identifying the distributed classes

The first step to building our remote event service is to identify exactly what classes need to be remote. In this case there are three:

* The **event service** must be remote so it can be instantiated in its own JVM and located in the network.
* The **event source** must be remote so that a reference to it can be included within the event data. A reference back to the source is often included to ensure that the listener can retrieve more data if the event data wasn't comprehensive enough.
* The **event listener** must be remote so that it can register with the event service and receive callbacks.

Notice that the event data object doesn't have to be a remote object. This is because the event data object will be serialized between processes and each process will have its own copy of the event data.

The following UML diagram illustrates the classes we'll use to construct the remote event service:



Modifying the classes

The necessary class modifications for this exercise are relatively trivial. They are as follows:

* All classes that will be remote objects will be split into an *interface* and an *implementation* class. The interface class must extend **`java.rmi.Remote`**. The implementation class must extend the new interface class and implement **`java.rmi.server.UnicastRemoteObject`**.

* All methods that will be part of the remote objects public interface must be rewritten to throw **`java.rmi.RemoteException`**.

* The single main program must be split into three parts: one for the temperature gauges, one for the event service, and one for the thermostat.

* The lookup of the event service must change from the Singleton static call to a lookup in

naming.

Code examples

As an example, let's look at the core portion of the main program for starting the event service. The following code creates the remote registry, then an instance of the event service, and lastly, it registers the event service with naming:

```
LocateRegistry.createRegistry(REGISTRY_PORT);
EventService es = EventServiceImpl.getEventService();
Naming.bind("EventService",es);
```

The thermostat and the temperature gauges look up the event service and then complete method calls on the event service, as shown here:

```
ThermostatImpl t = new ThermostatImpl();
EventService remoteService = null;
String rmiServer =
  "rmi://"+args[0]+":"+StartEventService.REGISTRY_PORT+"/EventService";
remoteService = (EventService)Naming.lookup(rmiServer);
remoteService.addTemperatureChangeListener(t);
```

The complete source for the remote example is in the accompanying source file (see Resources on page 30). Given our evolutionary approach to building a stand-alone event service, you should find that you understand the code very well at this point.

Problem solving the distributed event service

While it is easy to build a first distributed event service, there are many problems that can occur when it comes to run time. The two most critical problems are:

* **Listeners** can crash, causing remote exceptions when the event service tries to call them.

* The **event service** can crash, which will result in all events that were in process at the time being lost, and all sources being unable to send new events to the listeners. Processes must be put in place to relocate an event service after it crashes and re-register with it.

There are many different ways to resolve these problems. A discussion of even a portion of them is beyond the scope this tutorial. See Resources on page 30 for a list of recommendations for where to get started.

Summary

The distributed event service is a natural evolution of a stand-alone event service. With only minor changes we can now facilitate event-driven communication between applications running on different nodes throughout a network. While it is easy to show the concept of the distributed event service, building one that is robust and ready for enterprise usage can be difficult.

The source code for this example is in the **`com.stereobeacon.events.remote`** package (see Resources on page 30). A series of separate JVMs must be started to run the program. The JVMs can be on separate physical nodes.

Follow these steps to run the example:

1. Start the event service: **`java -classpath events.jar com.stereobeacon.events.remote.StartEventService`**.

2. Start the temperature gauges, being sure you place your machine name or IP Address as a parameter: **`java -classpath events.jar com.stereobeacon.events.remote.StartUpstairs MACHINENAME`**.

3. Start the thermostat, being sure you place your machine name or IP Address as a parameter: **`java -classpath events.jar com.stereobeacon.events.remote.StartThermostat MACHINENAME`**.

Section 6. Topic-based event delivery

Overview

Throughout this tutorial, we've changed the granularity of the notification from an object, to a property on an object instance, to an event on an instance, and finally to an event. We'll end the tutorial with a look at another type of event system entirely. Topic-based event delivery is in pervasive use in management applications and protocols, such as the Simple Network Management Protocol (SNMP) and the Jiro technology from Sun Microsystems. A *topic* is simply a string that denotes a branch of event types.

An example of topic-based event services is the Jiro technology event service. The event service uses point-delimited event hierarchies, with point (".") being the root event, and a tree of events being below the root. So, if ".filesystem" is an event, ".filesystem.outofspace" would be a more specific event. Listeners can register for any branch of a tree, and they will receive updates pertaining to that level and all events below it. Registering for ".filesystem" would deliver all events for the branch ".filesystem", such as ".filesystem.outofspace" or ".filesystem.fragmented".

Changes to the event service

To switch our event service to a topic-based model, we'll make big changes to its registration methods, as well as to how the event service manages subscriptions. This example supports only exactly matched topics; a more complex topic-matching exercise is left up to you.

In the code below you see a simple hashtable mechanism keyed on an event topic. The object associated with each key is a vector of listeners for the topic. Upon having an event fired, the hashtable is queried for listeners and the event is sent to each one, as shown below:

```
public class EventServiceImpl extends UnicastRemoteObject
      implements EventService
{
      // only relevant code shown...
      private Hashtable listeners = new Hashtable(1);
      public synchronized void addListener(String topic,
          TopicEventListener listener)
          throws RemoteException
      {
          if(debug) System.out.println("EventServiceImpl::addListener "+topic);
          Vector list = null;
          if(!listeners.containsKey(topic)){
              list = new Vector(1);
          } else {
              list = (Vector)listeners.get(topic);
              listeners.remove(topic);
          }
          list.add(listener);
          listeners.put(topic, list);
      }
      public synchronized void removeListener(String topic,
          TopicEventListener listener)
          throws RemoteException
      {
          Vector list = null;
```

```
            if(listeners.containsKey(topic)){
                list = (Vector)listeners.get(topic);
                listeners.remove(topic);
                list.remove(listener);
                if(list.size()>0)
                    listeners.put(topic, list);
            }
        }
        public void fireEvent(TopicEvent event)
            throws RemoteException
        {
            if(debug) System.out.println(
                    "EventServiceImpl::fireEvent "+event.getTopic());
            Vector list = null;
            if(listeners.containsKey(event.getTopic())){
                list = (Vector)listeners.get(event.getTopic());
                Enumeration e = list.elements();
                while(e.hasMoreElements()){
                    TopicEventListener tel = (TopicEventListener)e.nextElement();
                    tel.notify(event);
                }
            }
        }
 }
```

Changes to the event listeners and data

Because all the listeners will register through a single subscription mechanism, they must all support a single generic listener interface and accept a single event data object. (Nothing prevents you from subclassing the event data object to provide more information, however.)

Rather than use subclasses, we embed a hashtable within the event data object, which then collects information to relay to the listeners. The complete event data object is shown here:

```
 public class TopicEvent extends java.util.EventObject {
        private String topic = null;
        private Hashtable data = null;
        public TopicEvent(Object source, String topic, Hashtable data){
            super(source);
            this.topic = topic;
            this.data = data;
        }
        public String getTopic() {
            return topic;
        }
        public Hashtable getData(){
            return data;
        }
 }
```

Changes to the Listener interface

For the final step, we must update the **`Listener`** interface to support a generic method with the generic event data object. **`ThermostatImpl`** will implement the **`TopicListener`** interface. The updated method is shown here:

```
    public void notify(TopicEvent event)  throws RemoteException {
        String topic = event.getTopic();
        if(topic.equals(TemperatureGauge.CHANGE_EVENT_TOPIC)){
            Hashtable data = event.getData();
            String location = (String)data.get(TemperatureGauge.LOCATION_KEY);
            int newTemp =((Integer)data.get(TemperatureGauge.NEWTEMP_KEY)).intValue();
            if(newTemp < 64 && !coldRooms.contains(location)){
                coldRooms.add(location);
                if(coldRooms.size()==1){
                    setHeater(true);
                }
            } else if(newTemp > 64 && coldRooms.contains(location)){
                coldRooms.remove(location);
                if(coldRooms.size()==0){
                    setHeater(false);
                }
            }
        }
    }
```

Summary

Topic-based event mechanisms shrink the size of the interface on the event service by making the event mechanisms more generic in nature. The granularity of the subscription is the tree branch.

This type of event mechanism is extremely popular in management systems. The hierarchical topic structure often mirrors the nature of management system events.

As you saw here, the implementation of a topic-based event system is very similar to that of the other types of event systems. Once again, it took only a few simple, evolutionary steps to reach our new desired mechanism. Unfortunately, this distributed event service suffers from the same problems as the last distributed event service, such as remote exceptions from listeners and the potential for "lost" events if a source, the service, or a listener crashes.

Section 7. Summary and resources

Summary

Throughout this tutorial, we've evolved an event-delivery technique to address a variety of granularities, as summarized below:

* **Object**: The Observer design pattern
* **Property**: JavaBeans Bound Property pattern
* **Event from an object instance**: JavaBeans Multicast Event pattern
* **Event in a system**: Stand-alone event service
* **Event in a network**: Distributed event service
* **Event branch**: Topic-based event service

The event granularity helps to decide which event type is useful for your particular application or system of applications.

Several of the event granularities, such as the JavaBeans Bound Property and Multicast Event patterns, are constructed with tools integration in mind. Other event mechanisms are useful for enterprise application integration, such as the distributed event service. The topic-based event service is tailored to a specific domain.

Throughout the tutorial, we increased the complexity of our **`IndoorTemperature`** example, altering the granularity with minimal changes to code. In practice, you will use several of the event mechanisms reviewed here. Be patient, decide what granularity of events you need, and look for existing literature and design patterns to guide your implementation efforts.

Advanced exercises

To increase your proficiency in developing event mechanisms, you may want to further practice implementing the design patterns we've used throughout the tutorial. In addition to the Observer and Singleton patterns, you may want to study the following event handling patterns:

* Reactor
* Proactor
* Asynchronous Completion Token
* Acceptor-Connector

You may also want to try a number of exercises to increase the robustness of the distributed event service. Specifically, you might want to:

* Add guaranteed delivery for as long as the event service runs

* Add persistence so that guaranteed delivery can occur through restarts of the event service

* Change the topic-based event service to allow listeners to register for topic branches (for example, subscribing to ".network" will deliver events for ".network.change", ".network.change.route", and other events in the ".network" branch)

* Explore the distributed event services for the Jini and the Jiro technology platforms in more detail

Resources

Downloads

* Download events.jar, the complete source for the examples used in this tutorial.

* The *Java 2 platform, Standard Edition* (http://java.sun.com/j2se/) is available from Sun Microsystems.

* *Netbeans* (http://www.netbeans.org) is an excellent development environment, providing complete support for the JavaBeans event mechanism.

* The *Jiro technology platform* (http://www.sun.com/jiro/) offers a great example of a topic-based event system.

* The *Jini technology platform* (http://www.sun.com/jini/) offers another window into a distributed event system.

* *VisualAge for Java* (http://www-4.ibm.com/software/ad/vajava/) is the award-winning IDE from IBM. It's also the only Java development environment that lets you create and manage Java programs that can scale from Windows NT to OS/390 application servers.

* VisualAge is optimized for the *WebSphere application server family* (http://www-3.ibm.com/software/ts/mqseries/).

* If you've taken this tutorial to beef up your EAI skill set, you may want to check out *CrossWorlds Software* (http://www.crossworlds.com/ibm/index.html), enterprise application integration and B2B integration solutions from IBM.

Articles and tutorials

* The article, "*Management Application Programming: Getting started with the FMA and Jiro*" (developerWorks, May 2001, http://www-106.ibm.com/developerworks/java/library/j-jiro/), is an introduction to the Jiro technology platform and the Federated Management Architecture. If you want to learn more about distributed application programming, this is a good place to start.

* Learn more about how RMI works in an event-based architecture. Frank Sommers's "*Activatable Jini services: Implement RMI activation*" (developerWorks, October 2000, http://www-106.ibm.com/developerworks/java/library/j-jini/), shows you how to use RMI activation to manage computational resources in a distributed network.

* Ken Nordby's three-part series is a thorough (if slightly dated) introduction to JavaBeans technology. Read the whole series (developerWorks, June to August 2000):
  * *Part 1: The history and goals of EJB architecture* (http://www-106.ibm.com/developerworks/java/library/what-are-ejbs/part1/)

    * *Part 2: The EJB programming model* (http://www-106.ibm.com/developerworks/java/library/what-are-ejbs/part2/)
    * *Part 3: Deploying and using EJB components* (http://www-106.ibm.com/developerworks/java/library/what-are-ejbs/part3/)

* The tutorial *Enterprise JavaBeans fundamentals* (developerWorks, March 2001, http://www-106.ibm.com/developerworks/education/r-jejbf.html) is a more hands-on intro to Enterprise JavaBeans, with particular attention to EJB components in distributed-computing scenarios.

* The *Java design patterns 101 tutorial* (developerWorks, January 2002, http://www-106.ibm.com/developerworks/education/r-jpatt.html) is an introduction to design patterns. Find out why patterns are useful and important for object-oriented design and development, and how patterns are documented, categorized, and cataloged. The tutorial includes examples of important patterns and implementations.

Recommended books

* Deitel, Deitel, Santry, *Advanced Java 2 Platform, How to Program* (http://vig.prenhall.com/catalog/academic/product/1,4096,0130895601,00.html), Prentice Hall, New Jersey, 2002.

* Gamma, Helm, Johnson, Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (http://cseng.aw.com/book/0,3828,0201633612,00.html), Addison-Wesley, 1995.

* Monday, Connor, *The Jiro Technology Programmer's Guide and Federated Management Architecture* (http://www.jiro.com/documentcenter/book/guide/), Addison-Wesley, 2001.

* Schmidt, Stal, Rohnert, Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects* (http://www.wiley.com/Corporate/Website/Objects/Products/0,9049,104671,00.html), Wiley, 2000.

Additional resources

* Learn more about the Java 2 platform, Jiro technology, Jini technology, JavaBeans, and more. Visit the *Java Developer Connection* (http://developer.java.sun.com/developer/).

* You'll find hundreds of articles about every aspect of Java programming in the IBM developerWorks *Java technology zone* (http://www-106.ibm.com/developerworks/java/).

* See the *developerWorks tutorials page* (http://www-105.ibm.com/developerworks/education.nsf/dw/java-onlinecourse-bytitle?OpenDocument& for a complete listing of free tutorials from developerWorks.

* See the *Guide to developer kits from IBM* (http://www-106.ibm.com/developerworks/java/library/i-tools.html) for a listing of the latest IBM developer toolsets.

Feedback

Please send us your feedback on this tutorial. We look forward to hearing from you!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.