Engineering Software Development in Java

Lecture Notes for ENCE 688R, Civil Information Systems

Spring Semester, 2016

Mark Austin,
Department of Civil and Enviromental Engineering,
University of Maryland,
College Park,
Maryland 20742, U.S.A.

Chapter 15

Behavior Modeling Abstractions and Software Design Patterns

15.1 Abstractions for Modeling System Behavior

By definition, behavior is the way in which a organism, organ, or substance acts, especially in response to a stimulus (Merriam Webster, 1981). For our purposes, system behavior defines:

> **... what a system will do in response to its external environment without referring to details on implementation (e.g., use of technologies).**

In other words, the system behavior viewpoint focuses on the logical structure of the input and output, and the transformation function(s) that converts inputs into outputs. Understanding the behavior of a system as a whole requires (Kronloff, 1993):

1. A knowledge of the individual parts and their behavior,
2. The interfaces between the parts,
3. The traffic that passes along the interfaces, and
4. The system environment.

Elements of Behavior. The elements of behavior are as follows:

1. Functions. These are discrete tasks (or activities) that accept inputs and transform them into outputs.

   Functions may be decomposed into sub-functions.

   Individual functions are incapable of describing behavior.
2. Inputs and Outputs.
3. Control Operators. Define the ordering of functions.

Real-world behavior often emanates from the controlled ordering of functions, which in turn, affect how inputs are transformed into outputs.

15.2 Viewpoints of Behavior: Control, Data and State

The behavior of nearly all systems can be described using one or more of the following three viewpoints:

Viewpoint 1: Control Flow

The control flow model applies when you don't know when data will arrive; often, however, time of arrival matters more than the value of data itself.

Synthesis and evaluation methods emphasize event/reaction relationships, response time, and priorities among events and processes.

Connectivity and ordering of functions is handled by a second view that also incorporates control of flow. We need to know:

1. What happens and in what order?
2. What are the corresponding inputs and outputs?

Control flow takes each step when another step (or steps) is complete, perhaps taking into account other conditions, but without regard to the availability of inputs.

Inputs are determined during the transitions between steps, but are not required to start a step. No restrictions are placed on how inputs are determined.

When to use Control Flow

Control flow emphasizes the sequencing of steps by requiring one step to finish before another starts (e.g., postal delivery person).

It de-emphasizes the calculation of inputs by using whatever information happens to be available when a step starts, according to some unrestricted algorithm.

This makes control flow more suitable for applications in which the exact sequence of steps is most important, as in business applications and computer software.

Viewpoint 2: Data Flow

The data flow model applies when data arrives in regular streams. Specification of models focuses on functional dependencies between input and output – data flow takes each step when other steps provide its inputs.

Inputs are determined by being passed directly from outputs of other steps. Synthesis and validation methods emphasize memory/time efficiency. All events and processes are treated equally.

When to use Data Flow

Data flow emphasizes the calculation of inputs by requiring the outputs of one step to be explicitly linked to inputs of another step or steps. It de-emphasizes the sequencing of steps, because the time at which

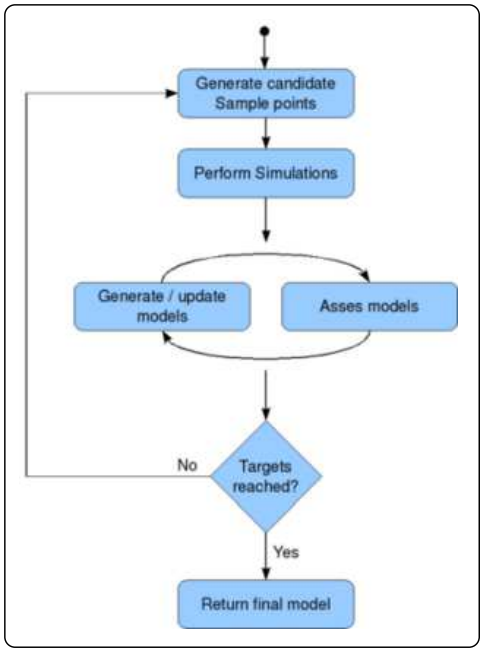


Figure 15.1. Control flow diagram for an iterative simulation process.

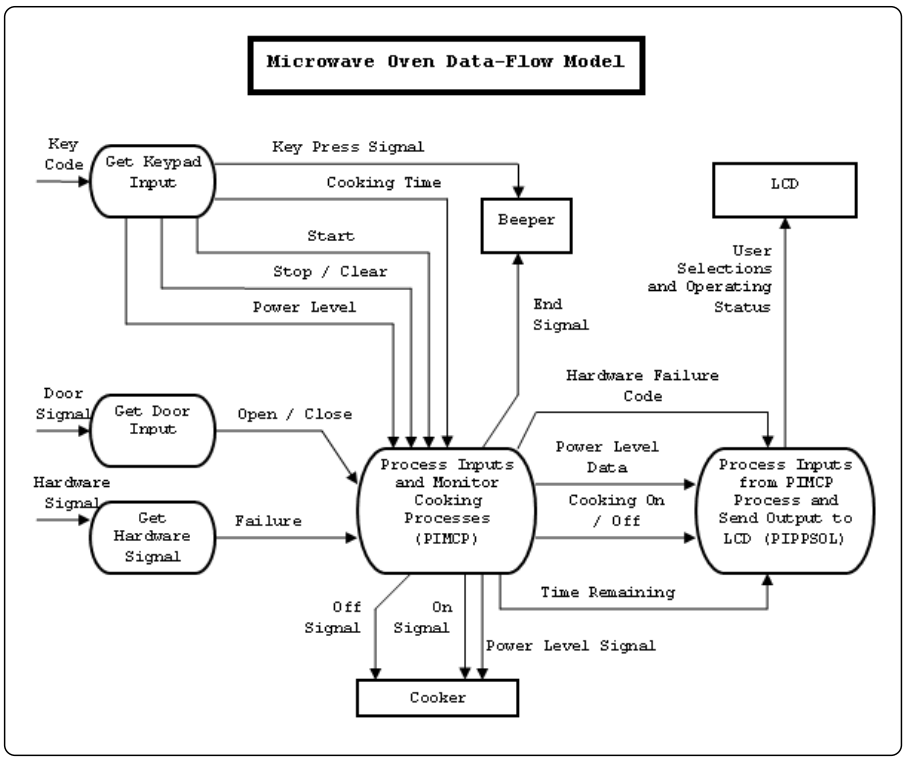


Figure 15.2. Data flow diagram for operation of a microwave oven.

all the inputs to a step arrive is determined by however long the various input steps take to complete. This means some steps may get all their inputs earlier than others in a less restrictive order than control flow.

These characteristics make data flow more suitable for applications in which the proper determination of inputs is most important, as in manufacturing applications, signal processing, and modeling of signal flows in hardware.

Viewpoint 3: State Machines

State machines emphasize response to external stimuli by requiring that each step start only when certain events happen in the environment of the system. The process by which events are generated and sequenced for consideration is not part of the state machine model. Events are recognized singly. Transitions fire one at a time. The state machine will only recognize those events defined in the model. The machine can only be in one state at a time. The current state cannot change except by a defined transition.

Suppose, for example, that with the passage of time, a system switches between states as shown in Figure 15.3.

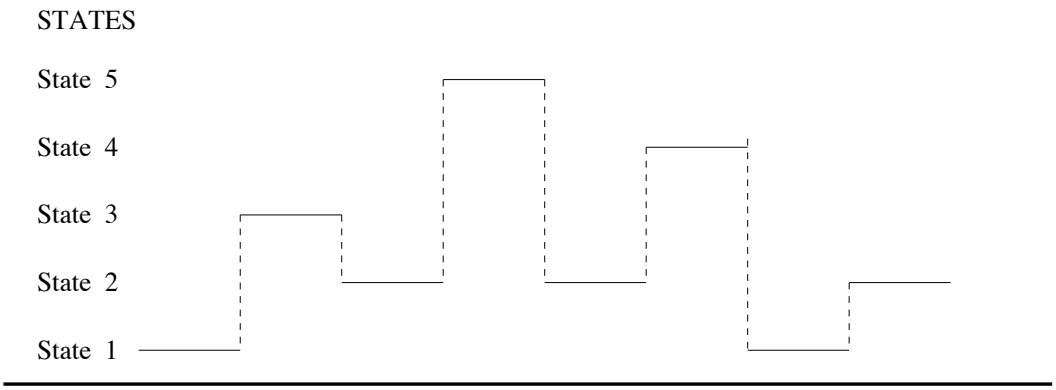


Figure 15.3. System state versus time.

A state machine is an abstraction composed of events (inputs), states, transitions, and actions (outputs):

1. **State.**
   An abstraction that summarizes that information concerning past inputs that is needed to determine the behavior of the system on subsequent inputs (Hopcroft 79, pg. 13);

2. **Transition.**
   A transition takes a system from one state to another (i.e., this is an allowable two-state sequence).

   A transition is caused by an event. It may result in an output action. It must specify an acceptable state and a resultant state;

3. **Event.**
   An input (or an interval of time);

There are four types of events that can trigger a state transition:

Type of Event	Action
Signal event	The system receives a signal from an external agent.
Call event	A system operation is invoked.
Timing event	A timeout occurs.
Change event	A system property is changed by an external agent.

Table 15.1. Types of events and associated actions.

4. The result (or output) that follows an event.

State machine mechanisms have several steps:

1. The machine begins at an initial state;

2. The machine waits for an event for an indefinite interval;

3. The event presents itself to the machine;

3. If the event is not accepted in the current state, it is ignored;

4. If the event is accepted in the current state, the designated transition is said to fire. The associated action (if any) is produced and the state designated as the resultant state becomes the current state. The current and resultant states may be identical.

5. The cycle is repeated from step 2, unless the resultant state is the final state.

State machines are static – that is, events and permissible states cannot be removed by execution of the machine. State machine models are time invariant (i.e., the firing of a transition in the presence of an event does not consume any specific amount of time).

Example: State Machine Behavior of a Spacecraft Computer System

Figure 15.4 (taken from Larson, 1996) shows a typical state transition diagram for a spacecraft computer system.

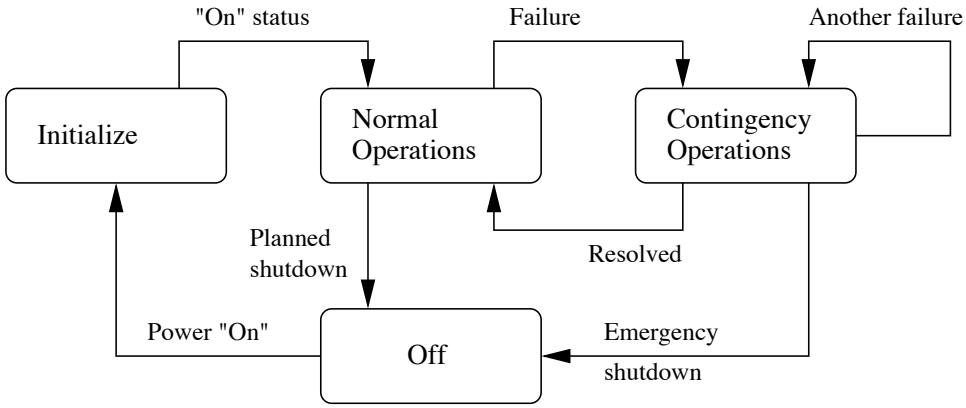


Figure 15.4. State machine model for behavior of a spacecraft computer.

Points to note:

1. The boxes in the state diagram show the valid states of the system, and the conditions needed to achieve each state.

2. The states Off and On are minimal choices for states.

   And even if the system is required to be On at all times, an Off should be provided for graceful shutdown in emergency situations.

3. The remaining states relate to what the system needs to do under normal and contingency operating conditions.

The computer system states must be consistent with the design requirements, and space crafts concept of correct operation.

When to use State Machines

State machine models are ideally suited to applications that operate in response to their environment, as in real-time or embedded applications.

The challenge for the designer is to devise a model that correctly represents the behavior of the system under study.

15.3 Mediator (Behavior) Design Pattern

Suppose that we want to design and implement a family of reusable components, but dependencies between the potentially reusable pieces are quickly evolving into a spaghetti code architecture. This situation can easily arise in programs that are made up of a large number of classes, and where the logic and computation is distributed among these classes. As classes are added to the program, the problem of communication among the classes becomes increasingly complex (i.e., tangled), which makes the program harder to read and maintain.

Mediator Design Pattern. The mediator design pattern is used to manage algorithms, relationships and responsibilities between objects.. It mitigates the aforementioned complexity problem by ...

... defining an object that controls how a set of objects will interact.

It allows loose coupling by encapsulating the way disparate sets of objects interact and communicate with each other. And it allows for the actions of each object set to vary independently of one another. Typical applications include mediation of messages among models and views, and control of activities for planes surrounding an airport.

To see why this pattern is useful, the left- and right-hand sides of Figure 15.5 show two approaches to connecting an ensemble of models to a collection of views. In the point-to-point solution, each model is connected to each view. For n models and n views this requires O(n^2) interfaces.

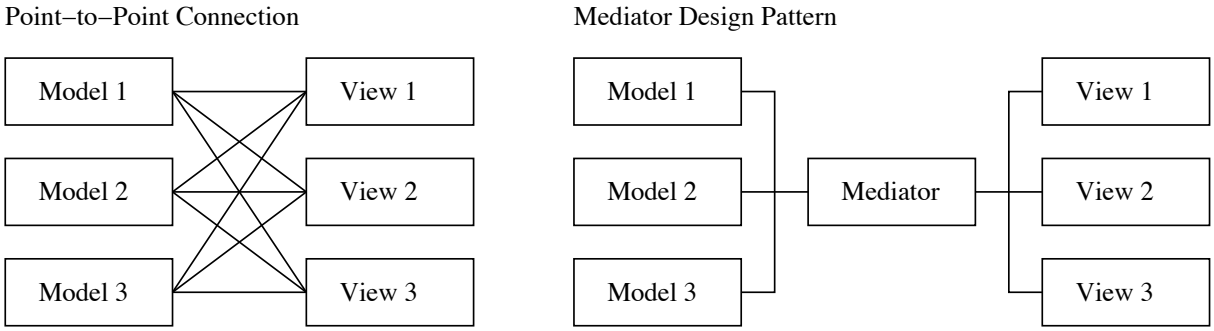


Figure 15.5. Communication between models and views. left: point-to-point connection, right: use of a mediator.

The mediator strategy of development simplifies communication between models and views because they do not need to implement the specific details of communication with each other. From a technical standpoint, the mediator object ...

... encapsulates all interconnections, acts as the hub of communication, and is responsible for controlling and coordinating the interactions of its clients.

Loose coupling between colleague objects is achieved by having colleagues communicate with the mediator, rather than with one another. It also provides maximum flexibility for expansion, because the logic for the communication is contained within the mediator.

Basic Implementation. A basic implementation of the mediator pattern has three parts:

1. **Mediator.** The mediator defines the interface for communication between colleague objects.
2. **ApplicationMediator.** The application mediator implements the methods defined in the mediator interface and coordinates communication among the colleague objects.

   During an actual implementation, the communication mediator will be aware of all the colleague clients, and their purpose with regards to inter-communication.
3. **ConcreteColleague.** The Concrete colleagues communicate with other colleagues through the mediator interface to the ApplicationMediator.

Figure 15.6 is a high-level schematic of the relationship among the mediator, concrete mediator and concrete colleague classes.

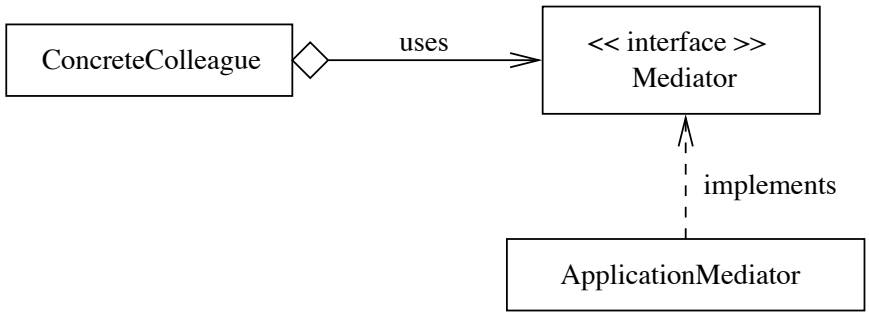


Figure 15.6. Basic relationship among classes in the mediator design pattern.

The Mediator class defines the interface for communication among colleague objects. The ConcreteMediator implements the Mediator interface and coordinates communication between Colleague objects. The ConcreteMediator keeps track of all the Colleagues and their purpose with regards to communication.

Example 01. Use a Mediator to Pass Messages among Digital Devices

Problem Statement. The purpose of this example is to demonstrate how the mediator design pattern can be used to manage that passing of message among models of desktop (e.g., iMac and PC) devices and mobile (e.g., iPhone and iPad) digital devices.

Figure 15.7 shows the relationship among classes for mediated passing of messages. To accommodate the two types of digital devices, we replace ConcreteColleague in Figure 15.6 with AbstractColleague, and then define DesktopColleague and MobileColleague as extensions of AbstractColleague.

The test program, DemoMediator, creates an instance of ApplicationMediator, two objects of type DesktopColleague (iMac and PC) and two objects of type MobileColleague (iPhone and iPad), and and the linkages among them to implement the mediator pattern. Communication among the colleagues is exercised in two parts:

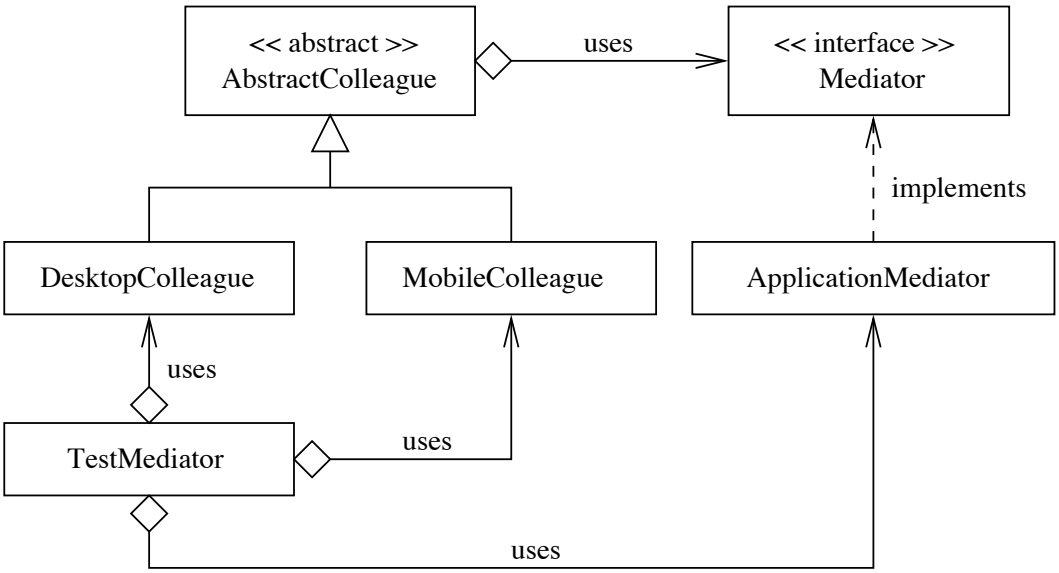


Figure 15.7. Relationship among classes for mediated passing of messages among digital devices.

Part 1. Desktop iMac sends the message: "Hello World from iMac ..."

Part 2. Mobile iPhone sends the message: "Hello from iPhone ..."

Complete scripts of the program I/O are given after explanation of the source code files.

File: mediator01/Mediator.java. This short file defines the mediator interface.

source code

```
/*
 *  ================================================================
 *  Mediator.java: Mediator interface ...
 *  ================================================================
 */

package mediator01;

public interface Mediator {
   public void send( String message, AbstractColleague colleague );
}
```

with a send(..) method that will contain a string message, and a reference to the source colleague – messages will be sent to all of the colleagues except the source.

File: mediator01/ApplicationMediator.java. The application mediator implements methods in the mediator interface and ...

source code

```
/*
 *  ================================================================
 *  ApplicationMediator.java:
 *  ================================================================
 */

package mediator01;

import java.util.ArrayList;

public class ApplicationMediator implements Mediator {
   private ArrayList<AbstractColleague> colleagues;

   public ApplicationMediator() {
      colleagues = new ArrayList<AbstractColleague>();
   }

   public void addColleague( AbstractColleague colleague ) {
      colleagues.add(colleague);
   }

   // Let all other screens know about a new message ...

   public void send( String message, AbstractColleague originator ) {
      for(AbstractColleague colleague: colleagues) {

          // No need to tell ourselves

          if( colleague != originator) {
              colleague.receive(message);
          }
      }
   }
}
```

uses an array list to store references to all of the registered colleagues. A hash set would also work. Notice that in the addColleague method, the method only refers to AbstractColleague (not the concrete implementations for Desktop and Mobile). The send() method calls the receive() methods in all of the registered colleages except the originating colleague (..no point in sending a message to yourself!).

File: mediator01/AbstractColleague.java. The AbstractColleague class contains a reference to the Mediator interface which, in turn, will be implemented as an instance of ApplicationMediator.

source code

```
/*
 *  ================================================================
 *  AbstractColleague.java: Methods for an abstract colleague. Notice
 *  that is uses the mediator interface.
 *  ================================================================
 */
```

```
package mediator01;

public abstract class AbstractColleague {
   private Mediator mediator;
   private String      sName;

   public AbstractColleague(Mediator m) {
      mediator = m;
   }

   // Set and get name ....

   public void setName ( String sName ) {
      this.sName = sName;
   }

   public String getName () {
      return sName;
   }

   // Send a message via the mediator

   public void send(String message) {
      mediator.send(message, this);
   }

   // Get access to the mediator

   public Mediator getMediator() {
      return mediator;
   }

   public abstract void receive(String message);
}
```

Since this class will be subclassed by both the Desktop and Mobile colleague representations, for clarity we include support for setting and getting the colleague names. A colleague will send a message to the application mediator through the method call:

```
mediator.send(message, this);
```

The reference to `this` is used to prevent a message being sent back to sender.

File: mediator01/DesktopColleague.java. The class DesktopColleague is a concrete extension of AbstractColleague.

source code

```
/*
 *  ================================================================
 *  DesktopColleague.java: Create a desktop colleague ...
```

```
 *  ================================================================
 */

package mediator01;

public class DesktopColleague extends AbstractColleague {
   public DesktopColleague ( Mediator m ) {
      super (m);
   }

   public void receive(String message) {
      System.out.printf("Desktop %s Received: %s \n", getName(), message);
   }
}
```

Notice that the constructor method makes an explicit reference to a constructor method in AbstactColleague.

File: mediator01/MobileColleague.java. The class MobileColleague is a concrete extension of AbstractColleague.

source code

```
/*
 *  ================================================================
 *  MobileColleague.java: Build a mobile colleague ...
 *  ================================================================
 */

package mediator01;

public class MobileColleague extends AbstractColleague {

   public MobileColleague ( Mediator m ) {
      super (m);
   }

   public void receive(String message) {
      System.out.printf("Mobile %s Received: %s \n", getName(), message);
   }
}
```

Again, notice that the constructor method makes an explicit reference to a constructor method in AbstactColleague.

File: mediator01/DemoMediator.java. The DemoMediator class builds and exercises the mediator application consisting of one application mediator, two desktop colleagues (iMac and PC), and two mobile colleagues (iPhone and iPad).

source code

```
/*
 *  ====================================================================
 *  DemoMediator.java: Build and exercise a mediator application.
 *
 *  Written by: Mark Austin                                 October 2012
 *  ====================================================================
 */

package mediator01;

public class DemoMediator {
   public static void main(String[] args) {

      // Create application mediator ....

      ApplicationMediator mediator = new ApplicationMediator();

      // Create colleagues ....

      AbstractColleague desktop01 = new DesktopColleague(mediator);
      desktop01.setName("iMac");
      AbstractColleague desktop02 = new DesktopColleague(mediator);
      desktop02.setName("PC");
      AbstractColleague  mobile01 = new MobileColleague(mediator);
      mobile01.setName("iPhone");
      AbstractColleague  mobile02 = new MobileColleague(mediator);
      mobile02.setName("iPad");

      // Register colleagues with the application mediator ..

      mediator.addColleague(desktop01);
      mediator.addColleague(desktop02);
      mediator.addColleague(mobile01);
      mediator.addColleague(mobile02);

      // Desktop 01 sends a message to his colleagues ...

      System.out.println("");
      System.out.println("Desktop iMac sends a message ... ");
      System.out.println("========================================");

      desktop01.send("Hello World from iMac ...");

      System.out.println("========================================");

      // Mobile iPhone sends a message to his colleagues ...

      System.out.println("");
      System.out.println("Mobile iPhone sends a message ... ");
      System.out.println("========================================");

      mobile01.send("Hello from iPhone ...");

      System.out.println("========================================");
```

```
  }
}
```

Program Output. In the fragment of program output:

```
prompt >>
prompt >> ant run13
Buildfile: /Users/austin/ence688r.d/java-code-design-patterns/build.xml

run13:
     [java]
     [java] Desktop iMac sends a message ...
     [java] ========================================
     [java] Desktop PC Received: Hello World from iMac ...
     [java] Mobile iPhone Received: Hello World from iMac ...
     [java] Mobile iPad Received: Hello World from iMac ...
     [java] ========================================
     [java]
     [java] Mobile iPhone sends a message ...
     [java] ========================================
     [java] Desktop iMac Received: Hello from iPhone ...
     [java] Desktop PC Received: Hello from iPhone ...
     [java] Mobile iPad Received: Hello from iPhone ...
     [java] ========================================

BUILD SUCCESSFUL
Total time: 3 seconds
prompt >>
```

the iMac desktop sends a message to his colleagues. Then in part 2, iPhone sends a message to his colleagues. In both cases, checks are made to ensure the message is not returned to sender.

15.4 Observer (Behavior) Design Pattern (and Listeners)

The observer pattern is applicable to problems where a message sender needs to broadcast a message to one or more receivers (or observers), but is not interested in a response or feedback from the observers.

Figure 15.8 shows the relationship of classes and interfaces in an implementation of the observer design pattern.

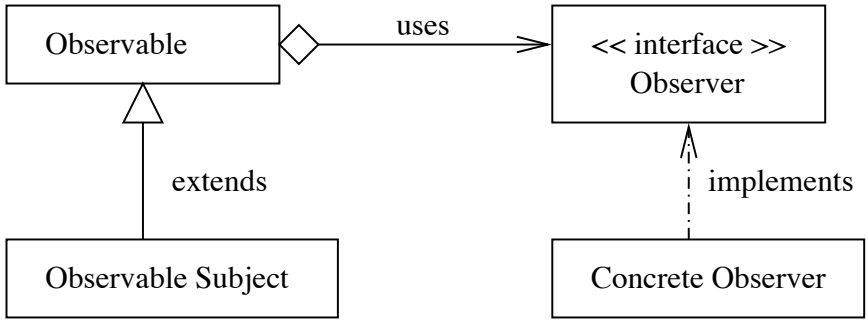


Figure 15.8. Relationship of classes and interfaces in the observer design pattern.

An observer will register for changes in the state of an observable object. When such a change occurs, the observer is notified and will handle the change through and event handler method.

Observer.java. Objects that implement the Observer interface will be will be informed of changes in an Observable.

```
package java.util;

public interface Observer {
   public void update(Observable observable, Object arg );
}
```

The update() method will be called whenever the observable object changes, and has called notifyObservers(). The Observable object can pass arbitrary information in the second parameter, i.e., the Observable object that changed and arbitrary information, usually relating to the change.

Observeable.java. Objects of type observable are observed. Other objects may register their intent to be notified when this object changes; and when this object does change, it will trigger the update() method of each observer. The abbreviated code is as follows:

```
package java.util;

public class Observable {
  private boolean         changed; // Track whether this object has changed ...
  private LinkedHashSet observers; // List of registered observers ...

  // Constructor method ...
```

```
  public Observable() {
     observers = new LinkedHashSet();
  }

  // Add an observer ....

  public synchronized void addObserver(Observer observer) {
     observers.add(observer);
  }

  // Return number of observers in this object ....

  public synchronized int countObservers() {
    return observers.size();
  }

  // Delete an observer from this Observable ....

  public synchronized void deleteObserver(Observer victim) {
     observers.remove(victim);
  }

  // Notify observers of a change ....

  public void notifyObservers() {
     notifyObservers(null);
  }

  // If object has changed, tell the observers about it ...

  public void notifyObservers(Object obj) {
     if (! hasChanged()) return;

     // Create clone inside monitor, as that is relatively fast and still
     // important to keep threadsafe, but update observers outside of the
     // lock since update() can call arbitrary code.

     Set s;
     synchronized (this) { s = (Set) observers.clone(); }
     int i = s.size();
     Iterator iter = s.iterator();
     while (--i >= 0)
         ((Observer) iter.next()).update(this, obj);
     clearChanged();
   }
}
```

Remark. The graphical user interface (GUI) event model used in Java Swing/AWT is based on the observer pattern. Graphical components (listeners) register for changes in other components by implementing the PropertyChangeListener interface. One can think of the event source component as a discrete observable, and the event change listener as a discrete observer.

Example 02. Use Observer Interface to implement a MessageBoard

Problem Statement. Figure 15.9 shows the relationship anong classes for a simple messageboard application.

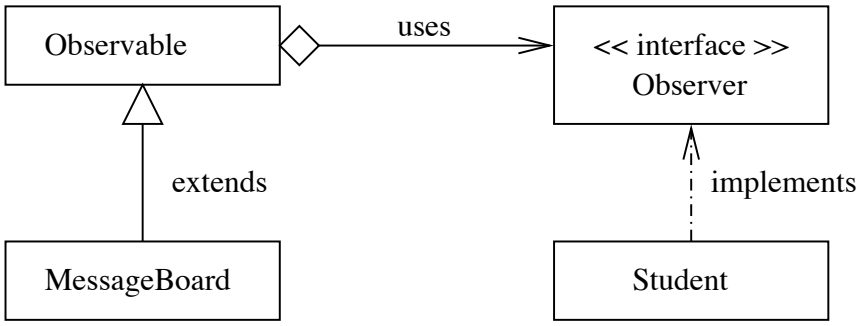


Figure 15.9. Relationship of classes and interfaces in the Student MessageBoard.

The MessageBoard class will be an extension of Observable. Students will be observers. The test program will simply create a message board and two student objects, and update the message on the board. The update will automatically forward the message to all students who are observing the board.

File: observer01/MessageBoard.java. The class MessageBoard is an extension of Observable.

source code

```
/*
 *  ================================================================
 *  MessageBoard.java: Create a simple message board ....
 *
 *  Written by: Mark Austin                                 June 2012
 *  ================================================================
 */

package observer01;

import java.util.Observable;
import java.util.Observer;

public class MessageBoard extends Observable {
   private String message;

   public String getMessage() {
      return message;
   }

   public void changeMessage(String message) {
      this.message = message;
      setChanged();
      notifyObservers(message);
   }
}
```

The method `changeMessage()` updates the message and then calls `notifyObservers()` to notify all observers.

File: observer01/Student.java. Our simple model of a student implements the observer interface (i.e., it provides details for an `update()` method.

source code

```
/*
 *  ====================================================================
 *  Student.java: A simple model of a student who listens for messages.
 *
 *  Written by: Mark Austin                                    June 2012
 *  ====================================================================
 */

package observer01;

import java.util.Observable;
import java.util.Observer;

public class Student implements Observer {
   private String sName;

   public Student ( String sName ) {
      this.sName = sName;
   }

   public void update( Observable o, Object arg ) {
      System.out.printf("Student: %s\n", sName  );
      System.out.printf("   Message board update: %s\n", (String) arg);
   }
}
```

For clarity of student object identification, we also provide methods to name students.

File: observer01/TestMessageBoard.java. The TestMessageBoard program create two student objects, Bob and Katie, and a simple message board.

source code

```
/*
 *  ====================================================================
 *  TestMessageBoard.java: Create students and a simple message board,
 *  then send message to the board. Make sure students read them ...
 *
 *  Written by: Mark Austin                                  June 2012
 *  ====================================================================
 */
```

```
package observer01;

public class TestMessageBoard {
   public static void main(String[] args) {
      MessageBoard board = new MessageBoard();

      Student bob   = new Student("Bob");
      Student katie = new Student("Katie");

      board.addObserver( bob   );
      board.addObserver( katie );

      board.changeMessage("More Homework!");
   }
}
```

Bob and Katie are registered as obsevers of the message board. Finally, a change of message "More Homework!" is sent to the message board.

Program Output. In the fragment of output:

```
prompt >> ant run08
Buildfile: /Users/austin/ence688r.d/java-code-design-patterns/build.xml

run08:
     [java] Student: Katie
     [java]    Message board update: More Homework!
     [java] Student: Bob
     [java]    Message board update: More Homework!

BUILD SUCCESSFUL
Total time: 3 seconds
prompt >>
```

Katie and Bob receive the message "More Homework!."

Example 03. Demonstrate that Observers work across Threads

Problem Statement. There are many real-world applications that involve concurrent behaviors and their interaction via observation. For example, cars approaching a traffic intersection will observe a traffic light for guidance on how to proceed.

Figure 15.10 shows the essential details of relationships among classes.

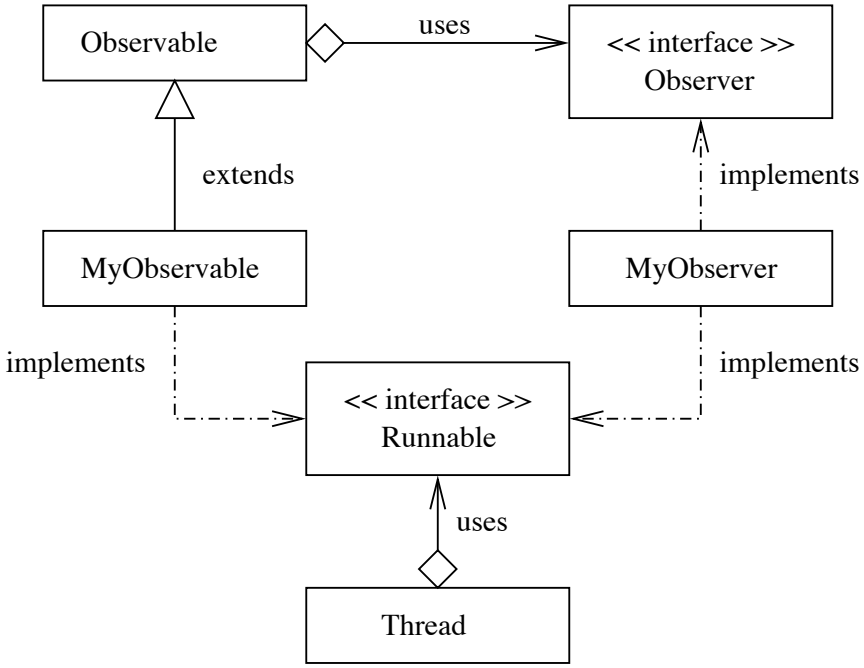


Figure 15.10. Relationship among classes in an application where observer objects are implemented as thread processes.

This program creates one observable process (MyObservable) and three observer processes (MyObserver). MyObservable increments a counter and its value is propogated to each of the registered observer processes.

MyObservable.java. The class MyObservable extends Observable (like the previous application) and implements the Runnable interface for executing a thread process.

source code

```
/*
 *  ================================================================
 *  MyObservable.java: Observer design pattern implemented as a thread.
 *  ================================================================
 */

package observer02;

import java.util.Observable;
```

```
import java.util.Observer;

public class MyObservable extends Observable implements Runnable {
  public MyObservable() {}

  public void start() { new Thread(this).start(); }

  public void run() {
    int count = 0;

    try {
      Thread.sleep(50);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    count++;

    System.out.println("*** In MyObservable: " + count );
    setChanged();
    notifyObservers( new Integer(count) );

    count++;

    System.out.println("*** In MyObservable: " + count );
    setChanged();
    notifyObservers( new Integer(count) );

    count++;

    System.out.println("*** In MyObservable: " + count );
    setChanged();
    notifyObservers( new Integer(count) );

    count++;

    System.out.println("*** In MyObservable: " + count );
    setChanged();
    notifyObservers( new Integer(count) );
  }
}
```

MyObserver.java.

source code

```
/*
 *  ===========================================================================
 *  MyObserver.java: Observer design pattern implemented as a thread ....
 *  ===========================================================================
 */

package observer02;
```

```
import java.util.Observable;
import java.util.Observer;

public class MyObserver implements Observer, Runnable {
   String name;

   public MyObserver() {}

   public MyObserver( String name ) {
      this.name = name;
   }

   public void start() {
      new Thread(this).start();
   }

   public void run() {}

   public void update(Observable o, Object arg) {
      Integer count = (Integer) arg;
      System.out.println("*** In observable thread:" + name + ":" + count.intValue());
   }
}
```

TestObserver.java.

source code

```
/*
 *  ================================================================================
 *  TestObserver.java: Observer design pattern implemented as a thread.
 *
 *  Written by: Mark Austin                                              June 2012
 *  ================================================================================
 */

package observer02;

import java.util.Observable;
import java.util.Observer;

public class TestObserver {
  public static void main(String[] args) {

    // Create a thread process that will be observed ...

    MyObservable observable = new MyObservable();

    // Create three processses that will observe MyObservable().

    MyObserver observer1 = new MyObserver("A1");
    MyObserver observer2 = new MyObserver("B1");
```

```
    MyObserver observer3 = new MyObserver("C1");

    // Register observers with observable process ...

    observable.addObserver(observer1);
    observable.addObserver(observer2);
    observable.addObserver(observer3);

    // Start observer processes ....

    observer1.start();
    observer2.start();
    observer3.start();

    // Start observable process .....

    observable.start();
    try {
      Thread.sleep(100);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Program Output. Insert material soon ....

```
prompt >> ant run09
Buildfile: /Users/austin/ence688r.d/java-code-design-patterns/build.xml

run09:
     [java] *** In MyObservable: 1
     [java] *** In observable thread:C1:1
     [java] *** In observable thread:B1:1
     [java] *** In observable thread:A1:1
     [java] *** In MyObservable: 2
     [java] *** In observable thread:C1:2
     [java] *** In observable thread:B1:2
     [java] *** In observable thread:A1:2
     [java] *** In MyObservable: 3
     [java] *** In observable thread:C1:3
     [java] *** In observable thread:B1:3
     [java] *** In observable thread:A1:3
     [java] *** In MyObservable: 4
     [java] *** In observable thread:C1:4
     [java] *** In observable thread:B1:4
     [java] *** In observable thread:A1:4

BUILD SUCCESSFUL
Total time: 3 seconds
prompt >>
```

Example 04. Use Listeners to Connect a Light and Light Watcher

Problem Statement. In this application we use listeners to connect a light and multiple light watchers. See Figure 15.11.

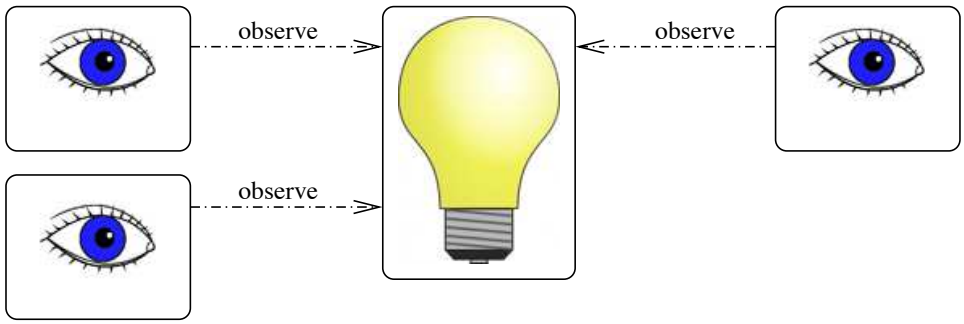


Figure 15.11. Schematic for a light and three light watchers (observers).

Each light watcher will listen for light events such as when the light is turned on and off.

Figure 15.12 ....

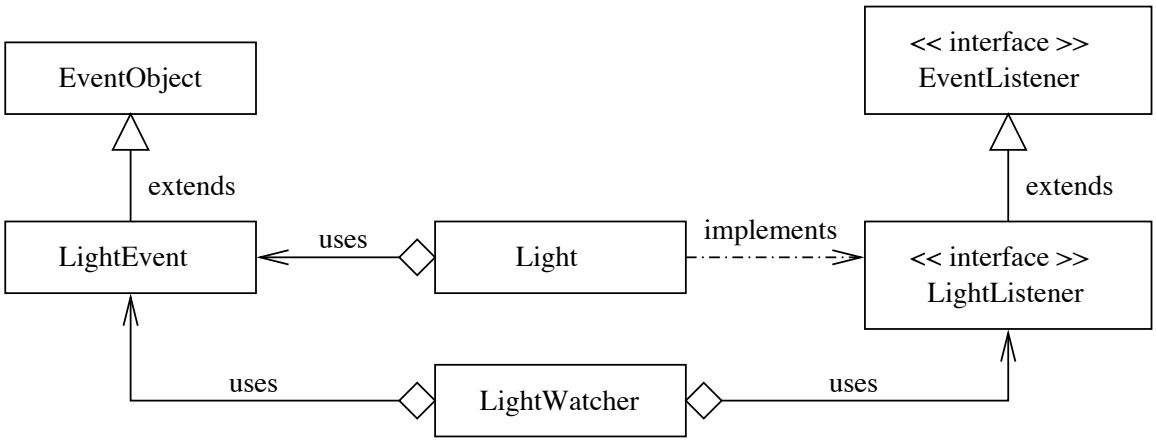


Figure 15.12. Relationship among classes for the light and light watchers (observers).

File. observer03/Light.java.

source code

```
/*
 *  ========================================================================
 *  Light.java: Simple model of a light source.
 *
 *  Modified by: Mark Austin                                    June 2012
 *  ========================================================================
*/
```

```
package observer03;

import java.util.*;

public class Light {
   private boolean on;   // on or off
   private List<LightListener> listeners = new ArrayList<LightListener>();

   // Light constructor ...

   public Light() {
      on = false;
      System.out.println("Light: constructed and off");
   }

   // Methods to add and remove listeners ...

   public void addLightListener(LightListener listener) {
      listeners.add(listener);
      System.out.printf("Light: Add LightWatcher %d\n",
                        ((LightWatcher) listener).id);
   }

   public void removeLightListener(LightListener listener) {
      listeners.remove(listener);
      System.out.printf("Light: Removed LightWatcher %d\n",
                        ((LightWatcher) listener).id);
   }

   // Methods to turn light on and off ....

   public void turnOn() {
      if (!on) {
         on = !on;
         System.out.println("Light: Turn on ... ");
         notifyListeners();
      }
   }

   public void turnOff() {
      if (on) {
         on = !on;
         System.out.println("Light: Turn off ... ");
         notifyListeners();
      }
   }

   // Notify watchers of the light ....

   private void notifyListeners() {
      LightEvent evt = new LightEvent(this);
      for (LightListener aListener : listeners) {
         if (on == true)
             aListener.lightTurnedOn(evt);
         else
```

```
            aListener.lightTurnedOff(evt);
      }
   }
}
```

File. observer03/LightListener.java.

source code

```
/*
 *  ========================================================
 *  LightListener.java: The LightListener interface
 *  ========================================================
 */

package observer03;

import java.util.EventListener;

public interface LightListener extends EventListener {
   public void lightTurnedOn(  LightEvent evt);
   public void lightTurnedOff( LightEvent evt);
}
```

File. observer03/LightEvent.java.

source code

```
/*
 *  ========================================================
 *  LightEvent.java ....
 *  ========================================================
 */

package observer03;

import java.util.EventObject;

public class LightEvent extends EventObject {
   public LightEvent (Object src) { super(src); }
}
```

File: observer03/LightWatcher.java.

source code

```
/*
```

```
 *  ========================================================================
 *  LightWatcher.java: A light watcher implements the LightListener interface.
 *  ========================================================================
 */

package observer03;

public class LightWatcher implements LightListener {
   int id;

   public LightWatcher (int id) {
      this.id = id;
      System.out.printf("Create LightWatcher %s\n", id );
   }

   // Implementation of event handlers

   public void lightTurnedOn(LightEvent evt) {
      System.out.printf("LightWatcher %d: I am notified that light is on\n", id );
   }

   public void lightTurnedOff(LightEvent evt) {
      System.out.printf("LightWatcher %d: I am notified that light is off\n", id );
   }
}
```

File: observer03/TestLight.java.

source code

```
/*
 *  ============================================================
 *  TestLight.java: Create one light and three light watchers ...
 *  ============================================================
 */

package observer03;

public class TestLight {
   public static void main(String[] args) {
      Light light = new Light();

      LightWatcher lw1 = new LightWatcher(1);
      LightWatcher lw2 = new LightWatcher(2);
      LightWatcher lw3 = new LightWatcher(3);

      // Lightwatchers lw1 and lw2 watch the light ...

      light.addLightListener(lw1);
      light.addLightListener(lw2);
      light.turnOn();

      // Lightwatchers lw3 watches the light ...
```

```
      light.addLightListener(lw3);
      light.turnOff();

      // Lightwatchers lw2 and lw3 no longer watch the light ...

      light.removeLightListener(lw2);
      light.removeLightListener(lw3);
      light.turnOn();
   }
}
```

Program Output. The script of code:

```
prompt >> ant run10
Buildfile: /Users/austin/ence688r.d/java-code-design-patterns/build.xml

run10:
     [java] Light: constructed and off
     [java] Create LightWatcher 1
     [java] Create LightWatcher 2
     [java] Create LightWatcher 3
     [java] Light: Add LightWatcher 1
     [java] Light: Add LightWatcher 2
     [java] Light: Turn on ...
     [java] LightWatcher 1: I am notified that light is on
     [java] LightWatcher 2: I am notified that light is on
     [java] Light: Add LightWatcher 3
     [java] Light: Turn off ...
     [java] LightWatcher 1: I am notified that light is off
     [java] LightWatcher 2: I am notified that light is off
     [java] LightWatcher 3: I am notified that light is off
     [java] Light: Removed LightWatcher 2
     [java] Light: Removed LightWatcher 3
     [java] Light: Turn on ...
     [java] LightWatcher 1: I am notified that light is on

BUILD SUCCESSFUL
Total time: 3 seconds
prompt >>
```

Example 05. Use Listeners to Model Node and Interval Dependencies

Problem Statement. In this example we use a network of listeners to model the dependency relationship between nodes and the intervals that are defined between them.

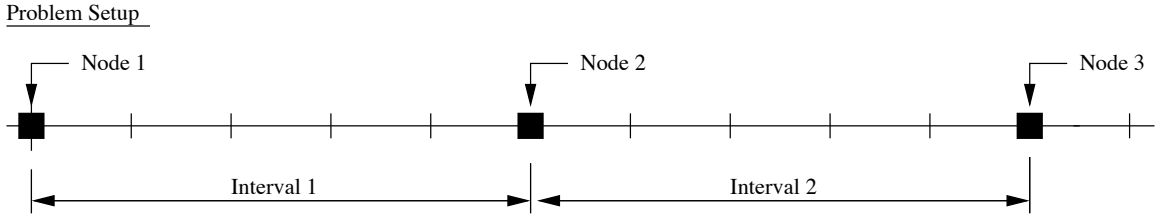


Figure 15.13. Problem setup: three nodes and two intervals.

To keep things simple, we only consider intervals defined by points positioned along a straight line. Figure 15.13 shows the problem setup. Three nodes are positioned at coordinates (0,0), (5,0) and (10,0). Interval 1 is defined by the space between nodes 1 and 2. Interval 2 is defined by the space between nodes 2 and 3.

A numerical experiment is conducted to test that the dependency relationship between the nodes and intervals works correctly: first, node 3 is moved one unit to the right. This increases the length of interval 2, but leaves interval 1 unchanged. Then, node 2 is moved one unit to the right. Interval 1 increases to length 6; interval 2 is decreased to length 5.

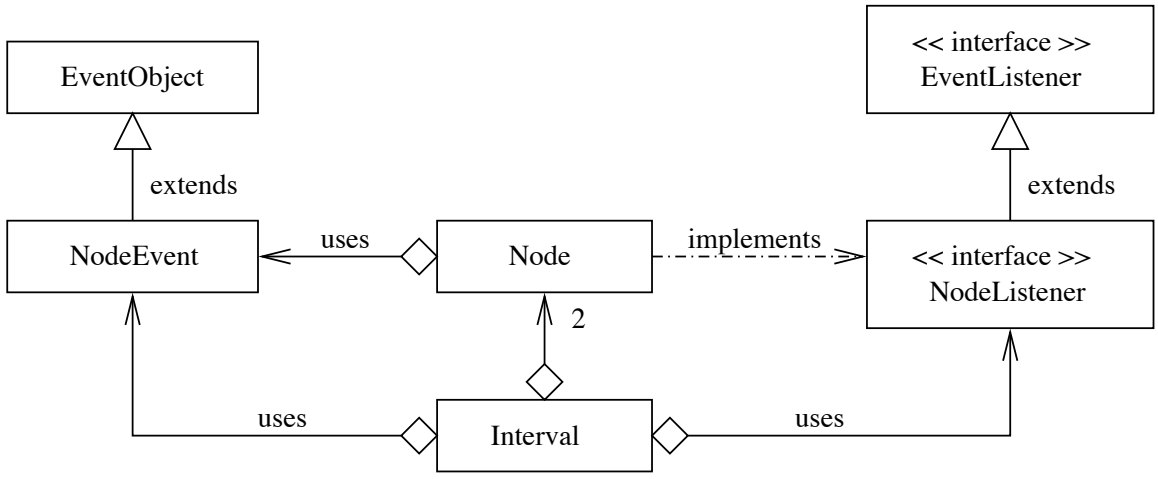


Figure 15.14. Relationship among classes for the node and interval dependency model.

Figure 15.14 illustrates the relationship among classes for the node and interval dependency model. Each interval is defined by the position of two nodes. Each interval also listens for node events, generated by a change in coordinate value. Each node keeps a list of registered listening objects. So, for example, node 1 has interval 1 as a registered listener. Node 2 has both intervals 1 and 2 as a registered listeners. Node 3 has interval 2 as a registered listener.

File: observer04/Node.java.

source code

```
/*
 *  ================================================================================
 *  Node.java: A simple node with x,y coordinates and support for Node Listeners
 *  ================================================================================
 */

package observer04;

import java.util.*;

public class Node {
   private List<NodeListener> listeners = new ArrayList<NodeListener>();
   private double x, y;

   // Node constructors ...

   public Node() {}

   public Node( double x, double y ) {
      this.x = x; this.y = y;
   }

   // Methods to set and get nodal coordinates ..

   public void setX( double x ) {
      this.x = x;
   }

   public void setY( double y ) {
      this.y = y;
   }

   public double getX() { return x; }
   public double getY() { return y; }

   // String description of the node ...

   public String toString() {
      String s = "Node: (x,y) = ( " + x + "," + y + " )";
      return s;
   }

   // Support for node listeners ...

   public void addNodeListener( NodeListener listener) {
      listeners.add(listener);
   }

   public void removeNodeListener( NodeListener listener) {
      listeners.remove(listener);
   }
```

```
   // Move node a distance dx, dy, then notify all registered listeners.

   public void move( double dx, double dy ) {
      if ( dx != 0.0 || dy != 0.0 ) {
         System.out.println("Node moved: ... ");
         x = x + dx;
         y = y + dy;
         notifyListeners();
      }
   }

   private void notifyListeners() {
      NodeEvent evt = new NodeEvent(this);
      for (NodeListener aListener : listeners) {
           aListener.nodeMoved(evt);
      }
   }
}
```

File: observer04/NodeEvent.java.

source code

```
/*
 *  ====================================================
 *  NodeEvent.java: ...
 *  ====================================================
 */

package observer04;

import java.util.EventObject;

public class NodeEvent extends EventObject {
   public NodeEvent ( Object src ) {
      super(src);
   }
}
```

File: observer04/NodeListener.java.

source code

```
/*
 *  ====================================================
 *  NodeListener.java: The NodeListener interface
 *  ====================================================
 */

package observer04;
```

```
import java.util.EventListener;

public interface NodeListener extends EventListener {
   public void nodeMoved( NodeEvent evt );  // callback when a node is moved ...
}
```

File: observer04/Interval.java.

source code

```
/*
 *  ============================================================
 *  Interval.java: An Interval listener class
 *  ============================================================
 */

package observer04;

import java.lang.Math;

public class Interval implements NodeListener {
   Node end1, end2;
   double length;
   int id;

   // Constructor methods ...

   public Interval (int id) {
      this.id = id;
      System.out.println("Interval-0" + id + ": created");
   }

   public Interval (int id, Node end1, Node end2 ) {
      this.id   =   id;
      this.end1 = end1;
      this.end2 = end2;
      setLength();
   }

   // Compute length of interval

   public void setLength() {
      length = Math.abs( end1.getX() - end2.getX() );
   }

   public double getLength() {
      return length;
   }

   // String description of the interval ...
```

```
   public String toString() {
      String s = "Interval: id = " + id + " length = " + length;
      return s;
   }

   // Implementation of event handlers

   public void nodeMoved( NodeEvent evt ) {
      setLength();
      System.out.println(" Interval-0" + id + ": New length = " + length );
   }
}
```

File: observer04/TestIntervalAssembly.java.

source code

```
/*
 *  ============================================================
 *  TestIntervalAssembly.java. Create a line of three nodes and
 *                             two intervals.
 *
 *  Written By: Mark Austin                              June 2012
 *  ============================================================
 */

package observer04;

public class TestIntervalAssembly {
   public static void main(String[] args) {

      // Create a line of three nodes ...

      Node node01 = new Node( 0.0, 0.0);
      Node node02 = new Node( 5.0, 0.0);
      Node node03 = new Node(10.0, 0.0);

      // Define intervals between nodes ...

      Interval int01 = new Interval(1, node01, node02 );
      Interval int02 = new Interval(2, node02, node03 );

      // Print details of the nodes and intervals ...

      System.out.println( "Nodes ... " );
      System.out.println( "=========================" );

      System.out.println( node01 );
      System.out.println( node02 );
      System.out.println( node03 );

      System.out.println( "Intervals ... " );
      System.out.println( "=========================" );
```

```
      System.out.println( int01 );
      System.out.println( int02 );

      // Register intervals as node event listeners ...

      node01.addNodeListener( int01 );
      node02.addNodeListener( int01 );
      node02.addNodeListener( int02 );
      node03.addNodeListener( int02 );

      // Move node 3 to coordinate (11,0).
      // This will only affect interval 2.

      node03.move( 1.0, 0.0 );

      // Move node 2 to coordinate (6,0).
      // This will affect intervals 1 and 2.

      node02.move( 1.0, 0.0 );
   }
}
```

Program Output.

```
prompt >> ant run11
Buildfile: /Users/austin/ence688r.d/java-code-design-patterns/build.xml

run11:
     [java] Nodes ...
     [java] ==========================
     [java] Node: (x,y) = ( 0.0,0.0 )
     [java] Node: (x,y) = ( 5.0,0.0 )
     [java] Node: (x,y) = ( 10.0,0.0 )
     [java] Intervals ...
     [java] ==========================
     [java] Interval: id = 1 length = 5.0
     [java] Interval: id = 2 length = 5.0
     [java] Node moved: ...
     [java]  Interval-02: New length = 6.0
     [java] Node moved: ...
     [java]  Interval-01: New length = 6.0
     [java]  Interval-02: New length = 5.0

BUILD SUCCESSFUL
Total time: 3 seconds
prompt >>
```

15.5 State Design Pattern

The state design pattern is used in software development to ...

... represent objects that will behave in different ways that depend on internal state.

The common features among the states can be represented by an interface class, and controled through the use of a context class. In this respect, the state design pattern is similar to the strategy design pattern.

Example 06. Simplified Modeling of a Finite State Machine

Purpose. To easily change an object's behavior at runtime.

Description. The state design patterns allows for the dynamic real-time adjustment of object behavior. It represents the states of an object as discrete objects.

Implementation. Dynamic behavior is achieved by delegating all method calls that certain values of to a State object (i.e., to the system's current state). In this way, the implementation of those methods can vary with the state of the object. Or in certain states some methods may not be supported at all.

Figure 15.15 shows the class diagram for the state design pattern.

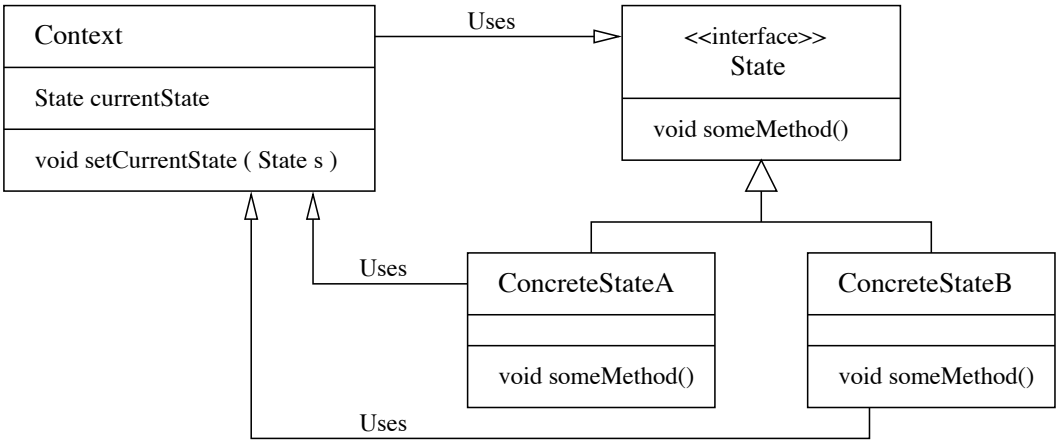


Figure 15.15. State class diagram (pg. 106 of Stelting).

Implementation of the state design pattern requires:

1. A `Context` object that keeps reference to the current state. It also acts as the interface for other clients to use. State-specific method calls are delegated to the current state object.

2. A `State` interface that defines all of the methods that depend on the state of the object.

3. A family of `ContreteState` objects. Each object will comply with the `State` interface and fill in details of system behavior for one specific system state.

Benefit. Use of the State class avoids the need for lengthy if-else/switch statements.

Example: Toggle Behavior for a Simple Button. Consider the behavior of button that can be simply toggled on and off. State behavior can be summarized as follows: (1) If the system state is ON and the button is pushed, then the system will transition to an OFF state, and (2) If the system state is OFF and the button is pushed, then the system will transition to an ON state.

The button object and its active object behavior are defined in the class files Button.java, State.java, ON.java and OFF.java. Button behavior is exercised via the ToggleButton class.

ToggleButton.java. Simulates the behavior by creating a (power) button object and then pushing it five times.

```
public class ToggleButton {
   public static void main( String[] args ) {
      Button power = new Button();
      for ( int i = 1; i <= 5; i = i + 1 )
         power.push();
   }
}
```

The output is as follows:

```
prompt >> java ToggleButton
   button: turning ON
   button: turning OFF
   button: turning ON
   button: turning OFF
   button: turning ON
prompt >>
```

State.java and **Button.java**. Button behavior is implemented through the use of a state design pattern – accordingly, the button class wraps the State interface.

```
public interface State {
   public void push( Button b );
}

public class Button {
   private State current;
   public Button()                  { current = OFF.instance(); }
   public void setCurrent( State s ) { current = s; }
   public void push()               { current.push( this ); }
}
```

ON.java and **OFF.java**. Concrete implementations of the button state are handled by the objects ON and OFF. Each object implements the State interface. For example, the fragments of code:

```
public class ON implements State {
   private static  ON inst = new ON();
   private ON() { }
```

```
   public static State instance() {
      return inst;
   }

   public void push( Button b ) {
      b.setCurrent( OFF.instance() );
      System.out.println( "   button: turning OFF" );
   }
}
```

defines the behavior for the state ON. When the button is pushed the behavior will transition to OFF.

15.6 Strategy Design Pattern

The strategy design pattern ...

> **... defines a family of algorithms, encapsulate each one, and make them interchangeable.**

The strategy pattern lets the algorithm vary independently from the clients that use it. This is achieved by capturing the abstraction of the algorithms in an interface and burying the implementation details in derived classes.

By coupling themselves to an interface (and not the implementation classes), client programs are ...

> **... isolated from changes to the number and details of the derived classes.**

This results in program modules that have high cohesion and low coupling, and interfaces that open extension, but closed for modifition (i.e., this is the so-called open/closed principle).

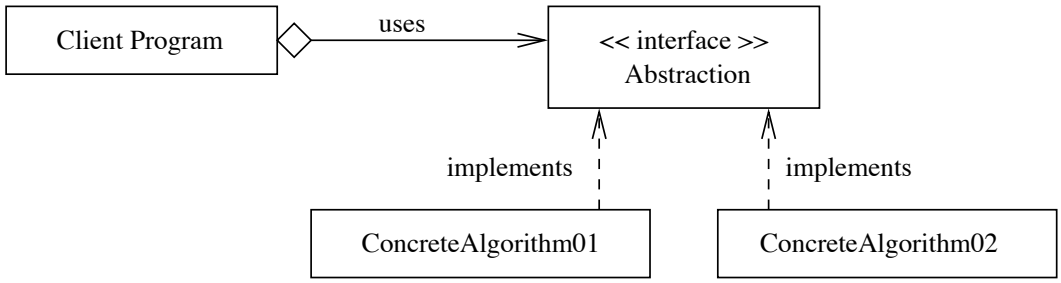


Figure 15.16. Class hierarchy for implementation of the strategy design pattern.

Figure 15.16 shows the relationship among classes for implementation of algorithms that are separated from a client by a well defined interface. Notice that the client program only uses the abstract interface and knows nothing about details of the concrete implementations. In other words, the client programs to an interface, not the details of an implementation.

Example 07. Run-Time Selection of Arithmetic Operations

This example demonstrates how the strategy design pattern can be used to create a program that allows for the run-time selection of different strategies (algorithms) for computing very simple arithmetic operations.

Figure 15.17 shows the class diagram for implementation of the strategy design pattern. From a practical standpoint we'd like to keep track of the implementation (algorithm) currently being used. A practical way of doing this is to introduce a context class between the client and the interface. The context will keep track of the current implementation in use. The client will have the ability to create a context and set the implemenation the needs to be executed.

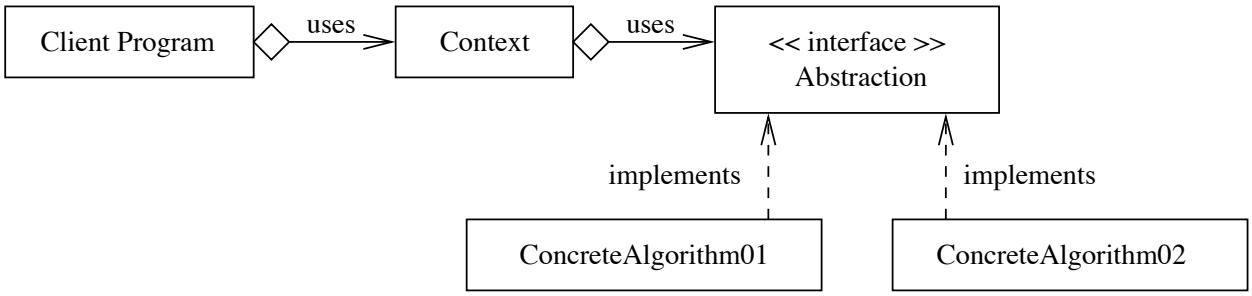


Figure 15.17. Class hierarchy for implementation of the strategy design pattern.

File: strategy01/Strategy.java This file defines methods in the strategy interface.

source code

```
/**
 *  ============================================================================
 *  Strategy.java: Concrete strategies will implement this interface.
 *  ============================================================================
 */

package strategy01;

public interface Strategy {
   int execute(int a, int b);
}
```

In this case, all communication between the implemented algorithms and the client program will occur through the execute() method.

File: strategy01/ConcreteStrategyAdd.java The details of a concrete implementation to add two integers are as follows:

source code

```
/**
 *  ============================================================================
 *  ConcreteStrategyAdd.java: Concrete strategy for adding integers.
 *  ============================================================================
 */

package strategy01;

public class ConcreteStrategyAdd implements Strategy {
   public int execute( int a, int b ) {
      System.out.println("*** In ConcreteStrategyAdd.execute() ...");
      return a + b;
   }
}
```

File: strategy01/Context.java. This file maintains a reference to the current strategy object. Notice that the context doesn't know anything about concrete strategy classes; it only knows about the strategy interface.

source code

```
/**
 *  ============================================================================
 *  Context.java: Maintain a reference to the current strategy object.
 *
 *  Notice that the context doesn't know anything about concrete Strategy
 *  classes, it only knows about the Strategy interface....
 *  ============================================================================
 */

package strategy01;

public class Context {
   private Strategy strategy;

   public void setAddStrategy() {
      this.strategy = new ConcreteStrategyAdd();
   }

   public void setMultiplyStrategy() {
      this.strategy = new ConcreteStrategyMultiply();
   }

   public void setSubtractStrategy() {
      this.strategy = new ConcreteStrategySubtract();
   }

   public int execute(int a, int b) {
      return strategy.execute(a, b);
   }
}
```

File: strategy01/RunPattern.java This file executes the strategy design pattern by systematically walking the program through three strategies: (1) AddStrategy() to add two integers, (2) MultiplyStrategy() to multiply two integers, and (3) SubStrategy() to subtract two integers.

source code

```
/**
 *  ============================================================================
 *  RunPattern.java: Exercise strategy design pattern example.
 *
 *  This example is a minor improvement of the Java Strategy Design Pattern
 *  example on wikipedia at: http://en.wikipedia.org/wiki/Strategy_pattern
```

```
 *  ================================================================================
 */

package strategy01;

public class RunPattern {
   public static void main(String[] args) {
      Context context;

      // Demonstration for selection of different strategies

      context = new Context();
      context.setAddStrategy();
      int resultA = context.execute(3, 4);

      context = new Context();
      context.setMultiplyStrategy();
      int resultB = context.execute(3, 4);

      context = new Context();
      context.setSubtractStrategy();
      int resultC = context.execute(3, 4);
   }
}
```

The abbreviated output is as follows:

```
prompt >> ant run12
Buildfile: /Users/austin/ence688r.d/java-code-design-patterns/build.xml

compile:
run12:
     [java] *** In ConcreteStrategyAdd.execute() ...
     [java] *** In ConcreteStrategyMultiply's execute()
     [java] *** In ConcreteStrategySubtract's execute()

BUILD SUCCESSFUL
Total time: 5 seconds
```

15.7 Visitor Design Pattern

The visitor design pattern is ...

> **... a way of separating an algorithm (system functionality) from an object structure on which it operates.**

The pattern should be used when you have ...

> **... distinct and unrelated operations to perform across a structure of objects.**

This strategy of software development allows for the creation of a data model with limited internal functionality and a set of visitors that perform unrelated operations upon the data. The visitors need not know details of the organizational structure beforehand. Further, new operations may be added to existing object structures without modifying those structures.

A Real-World Example. Most people rely on various forms of help to maintain their house. To keep the lawn and gardens tidy, a gardening service might visit the house, tend the gardens and mow the lawns. When a drain becomes blocked, a plumber will visit your house, diagnose the problem, and unblock the pipes. The landscaper and plumber are visitors that operate on your house in a distinct and specific way.

Applicability. The visitor pattern is used when:

1. Similar operations have to be performed on objects of different types grouped in a structure (a collection or a more complex structure).
2. There are many distinct and unrelated operations needed to be performed. Visitor pattern allows us to create a separate visitor concrete class for each type of operation and to separate this operation implementation from the objects structure.
3. The object structure is not likely to be changed but is very probable to have new operations which have to be added. Since the pattern separates the visitor (representing operations, algorithms, behaviors) from the object structure it's very easy to add new visitors as long as the structure remains unchanged.

Class Structure. Figure 15.19 shows the organization of classes and interfaces in a generic implementation of the visitor design pattern. The left-hand side defines the object structure for the application that will be visited. The right-hand side defines the visitor interface and concrete visitors (e.g., a print visitor, a drawing visitor). A single visitor object is used to visit all elements of the data structure.

The core of this pattern is the Visitor interface. This interface defines a visit() operation for each type of ConcreteElement in the object structure. Meanwhile, the ConcreteVisitor implements the operations defined in the Visitor interface. The concrete visitor will store local state between calls to individual data objects. typically as it traverses the set of elements. The element interface simply defines an accept() method to allow the visitor to run some action over that element - the ConcreteElement will implement this accept() method, i.e.,

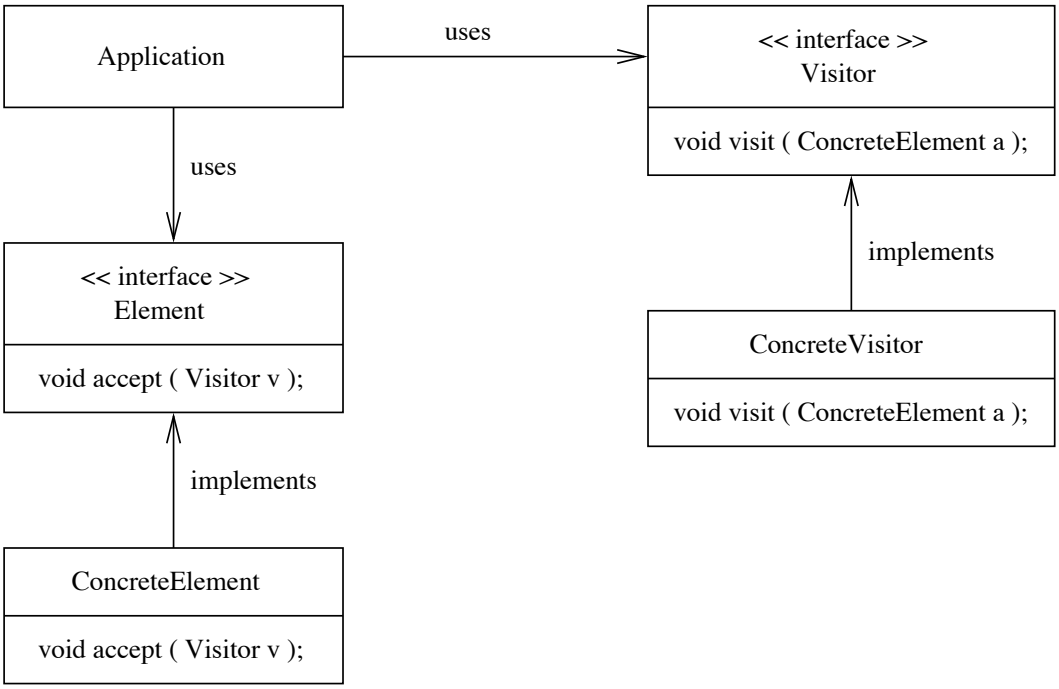


Figure 15.18. Organization of classes and interfaces in a generic implementation of the visitor design pattern.

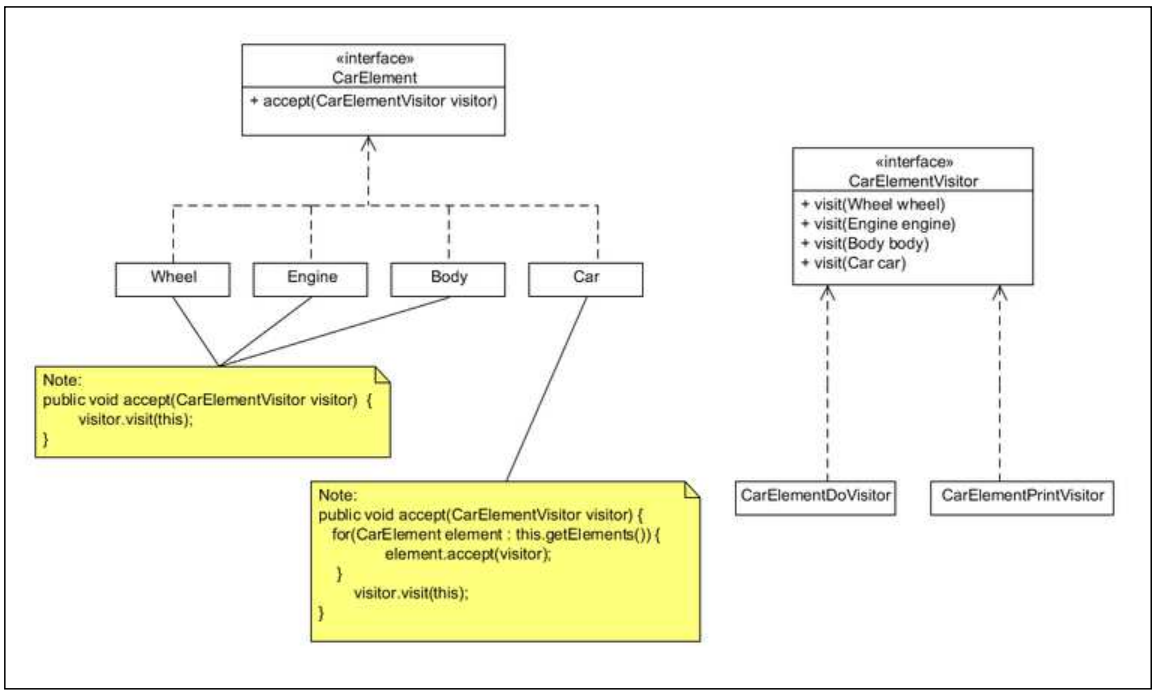


Figure 15.19. Class hierarchy for car model and visitor classes.

```
   public void accept( ConcreteVisitor visitor ) {
      visitor.visit(this);
   }
```

The accept() method performs a callback to the visitor, passing itself (i.e., an instance of ConcreteElement) to the visitor's method as a parameter.

To add a new operation, a new visitor class is created with the appropriate callback method. The data classes need no further modification.

Example 08. Visiting a Car Model

This example demonstrates the implementation of the visitor design pattern for a composite Car Model containing four wheels, a body and engine, and three visitor classes: (1) a draw visitor, (2) a print visitor, and (3) a price visitor. Figure 15.19 shows the class hierarchy for car model and visitor classes.

The Car Model

Body.java. Insert material soon ....

source code

```
/*
 *  ====================================================================
 *  Body.java: Specification for a car body ....
 *
 *  Modified by: Mark Austin                                  June, 2012
 *  ====================================================================
 */

public class Body implements CarElement {
   private double price;

   // Set/get methods for body price ....

   public void setPrice ( double price ) {
      this.price = price;
   }

   public double getPrice () {
      return price;
   }

   // Accept method for CarElement visitor ...

   public void accept( CarElementVisitor visitor ) {
      visitor.visit(this);
   }
}
```

Car.java. Insert material soon ....

source code

```
/*
 *  ====================================================================
 *  Car.java: Composite specification for a car ....
 *
 *  Modified by: Mark Austin                                  June, 2012
 *  ====================================================================
 */

public class Car implements CarElement {
    CarElement[] elements;

    // Car constructor methods ....

    public Car() {

        // Initialize details of car elements ...

        Wheel wheel01 = new Wheel("front left");
        wheel01.setPrice ( 1.0 );
        Wheel wheel02 = new Wheel("front right");
        wheel02.setPrice ( 1.0 );
        Wheel wheel03 = new Wheel("rear left");
        wheel03.setPrice ( 1.5 );
        Wheel wheel04 = new Wheel("rear left");
        wheel04.setPrice ( 1.5 );
        Body body    = new Body();
        body.setPrice ( 5.0 );
        Engine engine  = new Engine();
        engine.setPrice ( 5.0 );

        // Create an array of car elements

        this.elements = new CarElement[] { wheel01, wheel02, wheel03,
                                           wheel04,    body,  engine };
    }

    // Accept method for visiting elements ....

    public void accept( CarElementVisitor visitor ) {

        // Systematically visit the car elements ....

        for(CarElement elem : elements) {
            elem.accept( visitor );
        }

        // Finally, visit the car itself ....

        visitor.visit(this);
    }
}
```

Engine.java. Insert material soon ....

source code

```
/*
 *  ====================================================================
 *  Engine.java: Specification for a car engine ....
 *
 *  Modified by: Mark Austin                                  June, 2012
 *  ====================================================================
 */

public class Engine implements CarElement {
   private double price;

   // Set/get methods for engine price ....

   public void setPrice ( double price ) {
      this.price = price;
   }

   public double getPrice () {
      return price;
   }

   // Accept method for CarElement visitor ...

   public void accept(CarElementVisitor visitor) {
      visitor.visit(this);
   }
}
```

Wheel.java. Insert material soon ....

source code

```
/*
 *  ====================================================================
 *  Wheel.java: Specification for a car wheel ....
 *
 *  Note: The method:
 *
 *     Wheel.accept( CarElementVisitor )
 *
 *  overrides CarElement.accept(CarElementVisitor), so the call to
 *  accept is bound at run time. This can be considered the first
 *  dispatch. The decision to call:
 *
 *     CarElementVisitor.visit(Wheel)
 *
 *  however, rather than the other 'visit' methods in CarElementVisitor,
 *  can be made during compile time since 'this' is known at compile
```

```
 *  time to be a Wheel. Moreover, each subclass of CarElementVisitor
 *  overrides the visit(Wheel), which is another decision that is
 *  made at run time. This can be considered the second dispatch.
 *
 *  Modified by: Mark Austin                                  June, 2012
 *  ================================================================
 */

public class Wheel implements CarElement {
   private String  name;
   private double price;

   // Constructor methods ....

   public Wheel(String name) {
      this.name = name;
   }

   // Set/get methods for body name and price ....

   public String getName() {
      return this.name;
   }

   public void setPrice ( double price ) {
      this.price = price;
   }

   public double getPrice () {
      return price;
   }

   // Accept method for CarElement visitor ...

   public void accept( CarElementVisitor visitor ) {
      visitor.visit( this );
   }
}
```

Visitor Design Pattern Interfaces

CarElementVisitor.java. Insert material soon ....

source code

```
/*
 *  ================================================================
 *  CarElementVisitor.java: Car element visitor interface.
 *
 *  Modified by: Mark Austin                                  June, 2012
 *  ================================================================
 */
```

```
public interface CarElementVisitor {
    void visit(  Wheel  wheel );
    void visit( Engine engine );
    void visit(   Body   body );
    void visit(    Car    car );
}
```

CarElement.java. Insert material soon ....

source code

```
/*
 *  ====================================================================
 *  CarElement.java: Car element interface ....
 *
 *  Modified by: Mark Austin                                  June, 2012
 *  ====================================================================
 */

public interface CarElement {
    void accept( CarElementVisitor visitor );
}
```

Visitor Models

CarDrawVisitor.java. Insert material soon ....

source code

```
/*
 *  ====================================================================
 *  CarDrawVisitor.java: Visit car elements and draw details.
 *
 *  Modified by: Mark Austin                                  June, 2012
 *  ====================================================================
 */

public class CarDrawVisitor implements CarElementVisitor {
    public void visit( Wheel wheel ) {
       System.out.println("Draw a wheel: " + wheel.getName() + " ..." );
    }

    public void visit( Engine engine ) {
       System.out.println("Draw the engine ...");
    }

    public void visit( Body body ) {
       System.out.println("Draw the body ...");
    }

    public void visit( Car car ) {
```

```
        System.out.println("Draw my car ...");
    }
}
```

CarPrintVisitor.java. Insert material soon ....

source code

```
/*
 *  ====================================================================
 *  CarPrintVisitor.java: Visit car elements and print details.
 *
 *  Modified by: Mark Austin                                  June, 2012
 *  ====================================================================
 */

public class CarPrintVisitor implements CarElementVisitor {
    public void visit( Wheel wheel ) {
        System.out.println("Print: visiting " + wheel.getName() + " wheel");
    }

    public void visit( Engine engine ) {
        System.out.println("Print: visiting engine");
    }

    public void visit( Body body ) {
        System.out.println("Print: visiting body");
    }

    public void visit( Car car ) {
        System.out.println("Print: visiting car");
    }
}
```

CarPriceVisitor.java. Insert material soon ....

source code

```
/*
 *  ====================================================================
 *  CarPriceVisitor.java: Visit car elements and compute price ...
 *
 *  Modified by: Mark Austin                                  June, 2012
 *  ====================================================================
 */

public class CarPriceVisitor implements CarElementVisitor {
   private double totalPrice;

   public void visit( Wheel wheel ) {
```

```
      System.out.printf("Price: wheel  = %6.2f\n", wheel.getPrice() );
      totalPrice += wheel.getPrice();
   }

   public void visit( Engine engine ) {
      System.out.printf("Price: engine = %6.2f\n", engine.getPrice() );
      totalPrice += engine.getPrice();
   }

   public void visit( Body body ) {
      System.out.printf("Price: body   = %6.2f\n", body.getPrice() );
      totalPrice += body.getPrice();
   }

   public void visit( Car car ) {}

   // Return the internal state ....

   public double getTotalPrice() {
      return totalPrice;
   }
}
```

Visiting the Car Model

TestVisitorPattern.java. Insert material soon ....

source code

```
/*
 *  ========================================================================
 *  TestVisitorPattern.java: Exercise the visitor design pattern ...
 *
 *  Modified by: Mark Austin                                      June, 2012
 *  ========================================================================
 */

public class TestVisitorPattern {
    public static void main(String[] args) {
        Car car = new Car();

        System.out.println("" );
        System.out.println("Visit Car: Print Elements    " );
        System.out.println("============================" );

        car.accept( new CarPrintVisitor() );

        System.out.println("" );
        System.out.println("Visit Car: Draw Elements     " );
        System.out.println("============================" );

        car.accept( new CarDrawVisitor() );
```

```
        // Compute and print total cost of the car ...

        System.out.println("" );
        System.out.println("Visit Car: Economic Analysis " );
        System.out.println("============================" );

        CarPriceVisitor cpv = new CarPriceVisitor();
        car.accept( cpv );

        System.out.println("============================" );
        System.out.printf( "Total cost    = %6.2f\n", cpv.getTotalPrice() );
        System.out.println("============================" );
    }
}
```

The script of program output is as follows:

```
Script started on Fri Jun 22 08:53:14 2012
prompt >> java TestVisitorPattern.

Visit Car: Print Elements
============================
Print: visiting front left wheel
Print: visiting front right wheel
Print: visiting rear left wheel
Print: visiting rear left wheel
Print: visiting body
Print: visiting engine
Print: visiting car

Visit Car: Draw Elements
============================
Draw a wheel: front left ...
Draw a wheel: front right ...
Draw a wheel: rear left ...
Draw a wheel: rear left ...
Draw the body ...
Draw the engine ...
Draw my car ...

Visit Car: Economic Analysis
============================
Price: wheel  =   1.00
Price: wheel  =   1.00
Price: wheel  =   1.50
Price: wheel  =   1.50
Price: body   =   5.00
Price: engine =   5.00
============================
Total cost    =  15.00
============================
prompt >>
prompt >> exit
```

```
Script done on Fri Jun 22 08:53:21 2012
```

15.8 Model-View-Controller Design Pattern

The model-view-controller (MVC) design pattern divides a subsystem into three logical parts the model, view and controller and offers a systematic approach for modifying each part.

In the most common implementation of this pattern (see, for example, the Java patterns in Stelting and Maasson, 2002), views register for their intent to be notified when changes to a model occur. Controllers register their interest in being notified of changes to a view. When a change occurs in the view, the view (graphical user interface) will query the model state and call the controller if the model needs to be modified. The controller then makes the modification. Finally the model notifies the view that an update is required, based on a on change in the model.

In the second approach to the implementation of MVC, the controller is positioned at the center of the pattern and the models and views communicate through the controller channels. For example, after a view has notified the controller of a user action, the controller will update the property in the model based upon that action. From other direction, the controller registers for the changes in the model and updates the view based on the notification triggered from the model.

Example 09. Plan, Table, and Tree Views of Bouncing Balls

Insert screendumps and text soon .... get from Parastoo's report ....

Insert material soon ....

source code

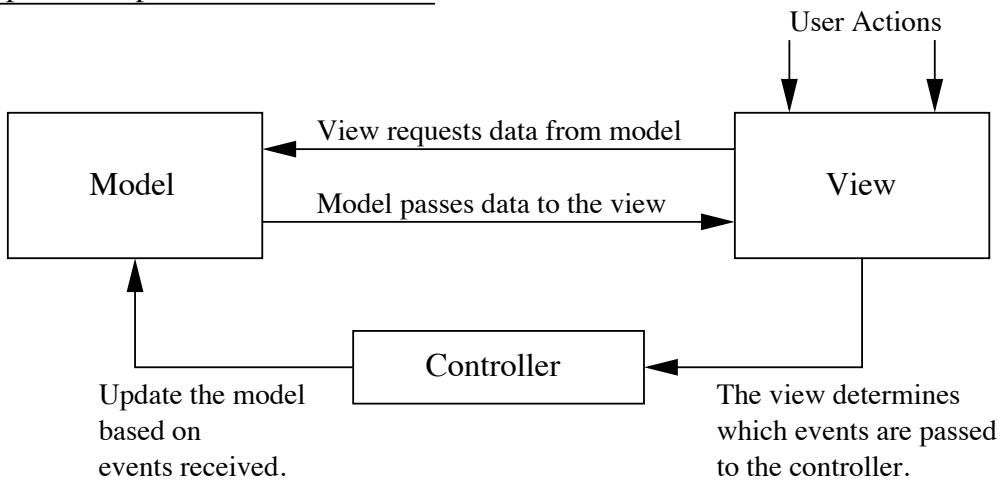
Example 10. Architectural Floorplan

Insert screendumps, text and source code describing the layout of building components – walls, doors, pillars, in a two-dimensional floorplan.

Insert material soon ....

source code

Simplified Implementation of MVC



Implementation of MVC with the Controller acting as a Mediator

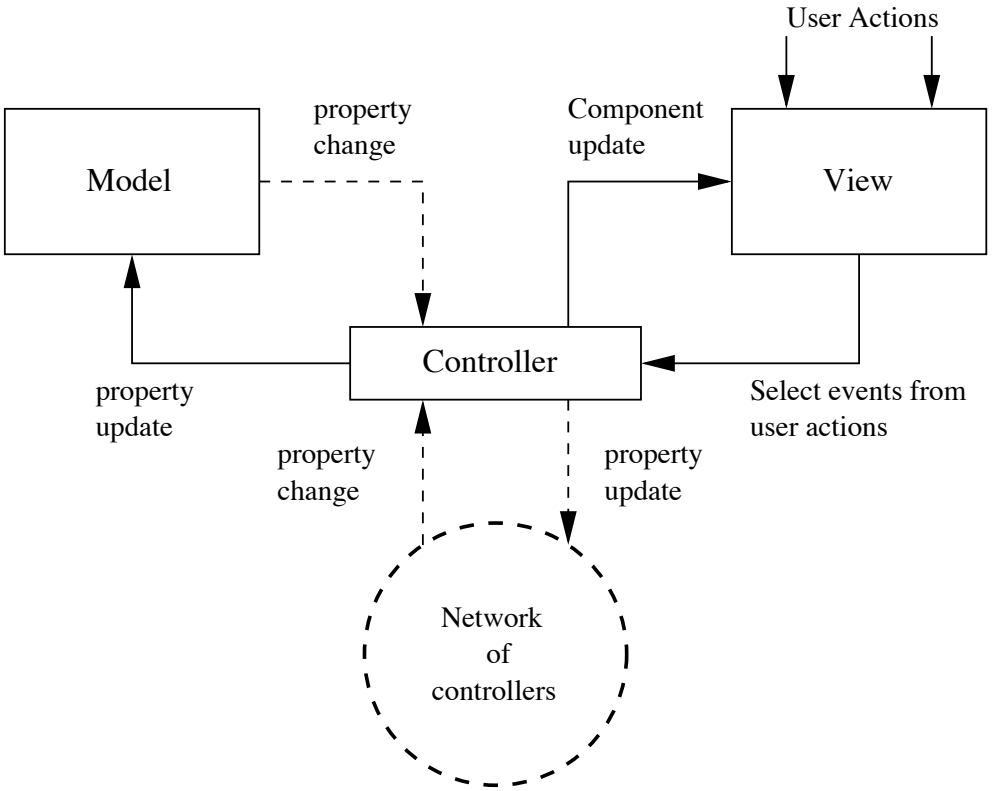


Figure 15.20. Styles of model-view-controller implementation.