The Design and Implementation of Multimedia Software

Supplemental Material

David Bernstein
James Madison University

Preface

The book *The Design and Implementation of Multimedia Software* assumes that the reader has a general understanding of object-oriented design and programming. This includes an understanding of the fundamentals of object-oriented programming (OOP) like classes and objects, abstraction, encapsulation, information hiding, the use of interfaces and abstract classes, specialization and inheritance, overriding and overloading, and polymorphism. This is standard material covered in most textbooks and undergraduate courses on object-oriented programming.

However, *The Design and Implementation of Multimedia Software* also assumes that reader has an understanding of some of the more common design patterns and experience with simple multi-threaded programming in Java. Since many readers will not have this background, and will not want to purchase books on each of these topics, this Supplement includes introductions to these two topics.

Contents

List of Figures

List of Tables

Multi-Threaded Programs

S1

Almost everyone that uses a computer today is, at least implicitly, familiar with the concept of *multi-tasking* because almost all modern operating systems allow users to "run more than one application at a time". The idea behind *multi-threading* is to allow each application to "perform more than one task at a time".

S1.1 Motivation

To see why multi-threading is important in some situations, consider an emergency vehicle dispatching system. In principle, such a system would allow an operator to track the current location of all emergency vehicles, dispatch emergency vehicles in response to "real-time" requests, and dispatch emergency vehicles on a regular schedule. This section only considers the dispatching functions and use the design shown in Figure S1.1.

The `Dispatcher` class maintains a collection of all available vehicles. In order that the vehicles be dispatched in a first-in-first-out manner, it uses a queue (of ID numbers) as follows:

```
import java.util.LinkedList;


public class Dispatcher
{
    protected int                     numberOfVehicles;
    protected LinkedList<Integer>     availableVehicles;
```

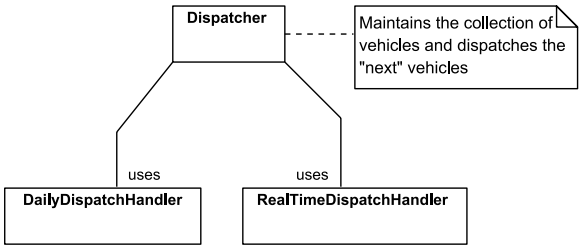

Figure S1.1 A Simple Emergency Vehicle Dispatching System

```
    public Dispatcher(int n)
    {
       int     i;

       numberOfVehicles = n;
       availableVehicles = new LinkedList<Integer>();

       for (i=0; i < n; i++)
       {
          makeVehicleAvailable(i);
       }
    }


}
```

When the `dispatch()` method is called, the next available vehicle is removed from the front of the queue and sent a message.

```
    public boolean dispatch(String task)
    {
       boolean  ok;
       int      vehicle;
       Integer  v;

       ok = false;
       v  = availableVehicles.removeFirst();

       if (v == null) ok = false;
       else
       {
          vehicle = v.intValue();
          sendMessage(vehicle, task);
          ok = true;
       }

       return ok;
    }
```

```
    private void sendMessage(int vehicle, String message)
    {

       // This method would normally transmit the message
       // to the vehicle.  For simplicity, it now writes it
       // to the screen instead.

       System.out.println(vehicle+"\t"+message+"\n");
       System.out.flush();
    }
```

When a vehicle completes its assigned task, its ID number is added to the end of the queue.

```
    public void makeVehicleAvailable(int vehicle)
    {
       availableVehicles.addLast(new Integer(vehicle));
    }
```

Now consider a `DailyDispatchHandler` that uses a `Dispatcher` object to handle a set of "regular" dispatches (i.e., dispatches that get handled in the same order every day) that are read in from a file. Each record in the "daily dispatch file" contains two fields, the task to be performed and the amount of time to wait before dispatching a vehicle to perform this task.

```
import java.io.*;
import java.util.*;

public class DailyDispatchHandler
{
    private Dispatcher  dispatcher;
    private String      fileName;

    public DailyDispatchHandler(Dispatcher d, String f)
    {
       dispatcher = d;
       fileName = f;
    }
```

```
}
```

The actual work is done by the `processDispatches()` method. This method reads each line form the file, tokenizes the line into its two fields, loops until the appropriate amount of time has elapsed, and then calls the `Dispatcher` object's `dispatch()` method.

```
public void processDispatches()
{
   BufferedReader    in;
   int               wait;
   long              currentTime, lastTime;
   String            line, message;
   StringTokenizer   st;

   try
   {
      in = new BufferedReader(new FileReader(fileName));

      lastTime = System.currentTimeMillis();

      while ((line = in.readLine()) != null)
      {
         st = new StringTokenizer(line,"\t");
         wait = Integer.parseInt(st.nextToken());
         message = st.nextToken();

         // Wait until the appropriate time before
         // dispatching this vehicle
         //
         while (System.currentTimeMillis()-lastTime
                < wait)
         {

            // Do nothing
         }

         dispatcher.dispatch(message);
         lastTime = System.currentTimeMillis();
      }
   }
```

```
        catch (IOException ioe)
        {
           System.out.println("No daily dispatches "+
                              "in: "+fileName);
        }
        catch (NoSuchElementException nsee)
        {
           System.out.println("Problem in file: "+fileName);
        }
    }
```

The dispatching process is started by a call to the `start()` method.

```
    public void start()
    {
       processDispatches();
    }
```

If you create a driver that constructs a `DailyDispatchHandler` and call its `start()` method you will see that it does what it is supposed to.

However, now suppose that, in addition to the `DailyDispatchHandlerer`, there is also a similarly implemented `RealTimeDispatchHandler`.[1]

```
import java.io.*;
import java.util.*;

public class RealTimeDispatchHandler
{
    private Dispatcher  dispatcher;

    public RealTimeDispatchHandler(Dispatcher d)
    {
       dispatcher = d;
    }
```

[1]Given the similarities between the `DailyDispatchHandlerer` and `RealTimeDispatchHandler` classes, one could have created a more generic class that would handle both types of dispatches. Two different classes are used because it makes the discussion that follows somewhat easier to understand.

```
    public void processDispatches()
    {
       BufferedReader    in;
       String            message;

       try
       {
          in = new BufferedReader(
                   new InputStreamReader(System.in));

          while ((message = in.readLine()) != null)
          {
             dispatcher.dispatch(message);
          }
       }
       catch (IOException ioe)
       {
          System.out.println("Problem with the console");
       }
    }

    public void start()
    {
       processDispatches();
    }

}
```

and execute the following driver:

```
import java.io.*;
import java.util.Date;

public class Driver
{
    public static void main(String[] args) throws IOException
    {
       BufferedReader           in;
       DailyDispatchHandler     daily;
       Dispatcher               dispatcher;
       RealTimeDispatchHandler  rt;
       String                   userInput;


       in=new BufferedReader(new InputStreamReader(System.in));
```

```
        dispatcher = new Dispatcher(50);


        daily = new DailyDispatchHandler(dispatcher,
                                         "dispatches.txt");
        daily.start();


        rt = new RealTimeDispatchHandler(dispatcher);
        rt.start();
    }

}
```

As the two classes above have been implemented, the `RealTimeDispatchHandler` can't start doing any work until the `DailyDispatchHandler` has completed its job. That is, the `while` loop in the `processDispatches()` method of the `DailyDispatchHandler` maintains control of the CPU until all of the daily dispatches have been completed.

What is needed is a way for multiple objects to use the `Dispatcher` "at the same time" (a phrase that is clarified in Section S1.3.1 on page 11). This can be accomplished using *multi-threading*.

S1.2 Getting Started with Threads

If you print out all of the classes in one of your applications on a big piece of paper (and you are a beginning programmer), you can trace its execution by drawing a single line connecting the statements in the order in which they are executed. This line can be thought of as a visual representation of the "thread of execution" for the application.

If multiple things happen "at the same time" in an application, then you can not trace its execution using a single line. Instead, you need to draw a separate line (each in its own color?) for each of the "threads of execution". If, at any point, two things happen "at the same time" then you need two line/threads, etc...

In Java you can make an application multi-threaded by taking advantage of the functionality of the `Thread` class. One way to do this is to create a class, for example, `ClassForApproachOne`, that extends the `Thread` class and put the code that should be executed in a separate thread of execution in the `run()` method. This is illustrated in Figure S1.2 on the following page. One must then create an instance of `ClassForApproachOne` and call it's (inherited) `start()` method. This causes the Java Virtual Machine to create a new thread of execution and call the `ClassForApproachOne` object's `run()` method in that thread of execution.

There are two problems with this approach, one conceptual and one practical. The conceptual problem is that it often leads people to believe that, because the specialization relationship is often called the "is a" relationship, the `ClassForApproachOne` object is a thread of execution. Unfortunately, while it "is a" `Thread` object, it is not a thread of execution. The

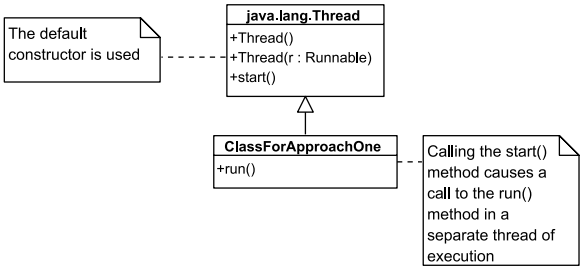

Figure S1.2 Specializing the `Thread` Class

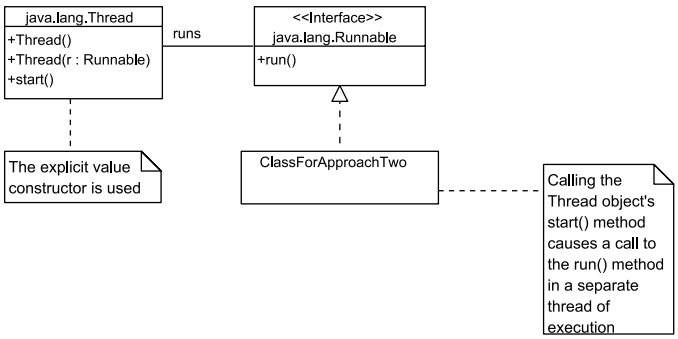

Figure S1.3 Implementing the `Runnable` Interface

practical problem is that, because Java does not support multiple inheritance, a class that extends `Thread` can't extend another class.

A better way to create a multi-threaded application is to create a class, for example, `ClassForApproachTwo`, that implements the `Runnable` interface as illustrated in Figure S1.3. Again, the code that should be executed in a separate thread of execution is put in the `run()` method. In this case, a `Thread` object must be constructed using the explicit value constructor, passing in the `Runnable` object. Then, when the `Thread` object's `start()` method is called, the `Runnable` object's `run()` method is called in the separate thread of execution.

One common way to use this approach is to include a `Thread` object as an attribute of the class that implements the `Runnable` interface. The thread of execution associated with the `Thread` object, and the `Thread` object itself, is then called the "control thread" for the `Runnable` object.

As an example, `DailyDispatchHandler` class is modified so that it does its work in its own thread of execution. The first thing to do is move the code from the `processDispatches()` method that should execute in a separate thread into the `run` method and indicate that the class now implements the `Runnable` interface.

```
import java.io.*;
import java.util.*;
```

```
public class DailyDispatchHandler implements Runnable
{
    private Dispatcher  dispatcher;
    private String      fileName;
    private Thread      controlThread;

    public DailyDispatchHandler(Dispatcher d, String f)
    {
       dispatcher = d;
       fileName = f;
    }



    public void run()
    {
       BufferedReader    in;
       int               wait;
       String            line, message;
       StringTokenizer   st;

       try
       {
          in = new BufferedReader(new FileReader(fileName));

          while ((line = in.readLine()) != null)
          {
             st = new StringTokenizer(line,"\t");
             wait = Integer.parseInt(st.nextToken());
             message = st.nextToken();

             try
             {
                // Sleep the appropriate amount of time
                // before dispatching the vehicle.  Other
                // threads can execute while this one
                // is sleeping.
                //
                controlThread.sleep(wait);
             }
             catch (InterruptedException ie)
             {
                // Do nothing
             }

             dispatcher.dispatch(message);
```

```
            }
         }
         catch (IOException ioe)
         {
            System.out.println("No daily dispatches "+
                               "in: "+fileName);

         }
         catch (NoSuchElementException nsee)
         {
            System.out.println("Problem in file: "+fileName);
         }
      }

}
```

Next, it is necessary to to change the `start()` method so that it creates a new thread of execution and processes the daily dispatches using this thread. This is accomplished by creating a `Thread` object and associating it with a `Runnable` object (in this case, `this DailyDispatchHandler` object). Then the `Thread` object's `start()` method is called.

```
    public void start()
    {
       if (controlThread == null)
       {
          controlThread = new Thread(this);
          controlThread.start();
       }
    }
```

Now when a `DailyDispatchHandler` object's `start()` method is called, the following things happen:

1. A `Thread` object named `controlThread` is constructed (causing a thread of execution to be created).
2. The `Thread` object's `start()` method is called (causing the thread of execution to be "started").
3. The `DailyDispatchHandler` object's `run()` method is called in the new thread of execution.

The `run()` method iteratively reads dispatch information from the file and calls the

`dispatch()` method until the end of stream indicator is reached. At this point, the thread of execution "drops out of" the `run()` method and, as discussed in Section S1.3.2 on the following page, it dies.

There is one other important change in this version of the `DailyDispatchHandler` class – the `run()` method now puts the `controlThread` object to sleep rather than looping until the appropriate amount of time has passed. This is an example of cooperative behavior. That is, rather than use the CPU needlessly, this approach allows other threads time to run on the CPU.

Now, because the `DailyDispatchHandler` object processes the dispatches file in a separate thread of execution, a `RealTimeDispatchHandler` can get some work done. In other words, one can now have two different objects dispatching vehicles "at the same time".

This can be demonstrated with the driver used earlier. In this case, the `RealTimeDispatchHandler` does its work in the "main" thread and the DailyDispatchHandler does its work in its own thread.

S1.3 Understanding Multi-Threading in Java

Before going any further, it is important to consider some of the details of the way the Java Virtual Machine manages threads.

S1.3.1 "At the Same Time"

Though it is common to use the phrase "at the same time" when talking about threads, it is important to understand that this phrase must be taken lightly. While some computers can execute more than one thread at a time (e.g., a computer with multiple CPUs, a computer with one CPU that has multiple cores[2]), many can't execute multiple threads simultaneously and yet can still support multi-threading. This is accomplished by executing the threads at almost the same time. In other words, the threads can take turns using the CPU.

Different operating systems handle mult-threading in different ways. Some use *time slicing* (sometimes called a *round robin* approach) in which the operating system allocates CPU time to threads. Others use a *cooperative* approach in which each thread controls how much CPU time it needs (hence a thread must yield control to other threads). MS-Windows uses a time slicing approach whereas most flavors of Unix/Linux support both approaches.

When writing multi-threaded programs in Java you should not make any assumptions about the underlying operating system or the number of CPUs. In particular, the Java scheduler does not implement time-slicing directly and, hence, does not guarantee it. Java *schedules* threads based on their *priority*. The scheduler chooses the runnable thread with the highest priority.

S1.3.2 The Thread Lifecycle in Java

As shown in the UML statechart diagram in Figure S1.4 on the next page a Java `Thread` object can be in one of four states, New Thread, Runnable, Not Runnable, or Dead. When a

[2] You can determine the number of available processors using the `availableProcessors()` method in the `RunTime` class.

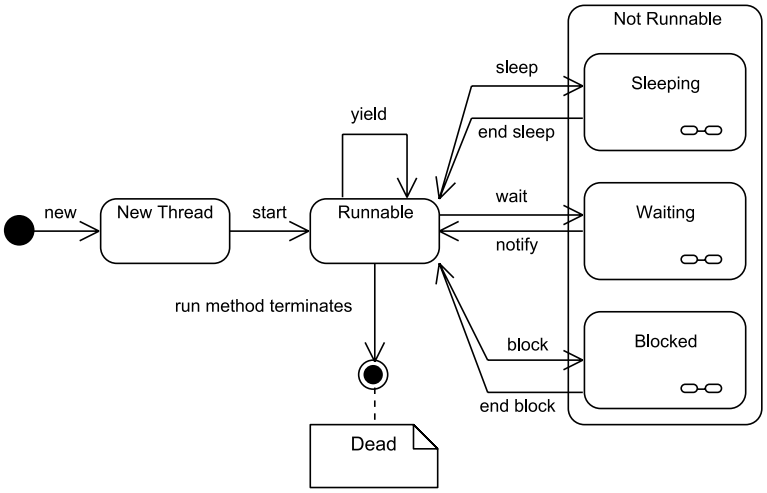

Figure S1.4 The Thread Lifecycle in Java

`Thread` is in the New Thread state, no system resources have been allocated to it, and none of its methods can be called except `start()`. The `start()` method allocates the necessary resources, schedules it to run, and calls the `run()` method. At this point the `Thread` is in the Runnable state. (It may or may not be "running" because it may be waiting for its turn on the CPU. For this reason, it is inappropriate to distinguish the "running" state from the "runnable" state.)

A `Thread` moves from the Runnable state to the Not Runnable state if its `sleep()` method is called, if it calls the `wait()` method, or if it is *blocking* (e.g., on input or output). The way a `Thread` goes from Not Runnable to Runnable depends on how it became Not Runnable – a call to `sleep()` causes it to remain Not Runnable for a pre-specified amount of time, if it is "waiting" then it transitions after it is "notified" (see page 21), and if it is blocking then it transitions when it stops blocking (e.g., after the input or output operation completes).

Note that a `sleep(duration)` call does not stop execution of the thread for exactly `duration` milliseconds, it stops execution for at least `duration` milliseconds. That is, the thread will not be scheduled for execution until `duration` milliseconds have elapsed. When it actually is executed will depend on the number and priority of other threads being scheduled.

Also note that every method that causes a thread to transition from Runnable to Not Runnable throws an `InterruptedException`, which is a *checked exception*. This enables the thread to respond to "abnormal requests" that it transition back to Runnable. In other words, this enables one thread of execution to call a Not Runnable thread of execution's `interrupt()` method, thereby requesting that it change state.

S1.3.3 Types of Threads

Every `Thread` object has a `setDaemon()` method that can be used to set the *daemon status* of its associated thread of execution. When a thread is created, it has the same daemon status as the thread that created it. If you want to change the daemon status, you must call the

`setDaemon()` method before you call the `start()` method.

A thread should be marked as a daemon only if it can be safely destroyed at any time (i.e., only if it is safe to stop executing code in that thread at any time). As a result, daemon threads should not be used frequently.

Daemon threads are normally used for background/helper activities. For example, the Java Virtual Machine runs the garbage collector in a daemon thread.

S1.3.4 Interruption

Every `Thread` object has an `interrupt()` method that can be used to set its *interrupt status* to `true`. This method is used to ask a thread to stop what it is currently doing. It is important to recognize that a thread need not be cooperative. That is, a call to the `interrupt()` method does not "cancel" the method being executed in a runnable thread, it simply changes the state of the interrupt status. The thread may or may not stop what it is doing as a result of this change of state.

Even in classes in the `java.lang` package, some methods act on this state change and some do not. For example, the `sleep()` method in the `Thread` class and the `wait()` method in the `Object` class both throw an `InterruptedException` (and clear the interrupt status) if the interrupt status is `true`.[3]

If you want to write a method that can be "cancelled" you need to include code that checks the interrupt status periodically using the `isInterrupted()` method in the `Thread` class. Obviously your documentation should include a description of when the method can be "cancelled" and how it responds.

S1.4 Tracing a Multi-Threaded Application

While it is nice, conceptually, to imagine multi-colored lines illustrating the execution order of a multi-threaded application, it is not very practical. A better approach is to write a "number" next to each line of code indicating the thread it is executed in and the order in which it is executed.

In the example below, lines in the *main* thread are numbered using 1,2,3..., lines in the second thread are numbered using A,B,C..., and line in the third thread are numbered using a,b,c... (For the purposes of this example, declarations of classes, variables, or methods are not treated as executable.)

```
public class SlasherDriver
{
    public static void main(String[] args)
    {
        Slasher     plus, slash;

1       slash = new Slasher();
6       slash.setCount(3);
8       slash.start();
```

[3]Since this is a checked exception, it must either be caught or specified (i.e., re-thrown).

```
10      plus = new Slasher("+");
14      plus.setCount(2);
16      plus.start();
    }
}


public class Slasher implements Runnable
{
    private int         count;
    private String      symbol;
    private Thread      controlThread;

    public Slasher()
    {
2       this("/");
    }

    public Slasher(String symbol)
    {
3 11    this.symbol = symbol;
4 12    count = 0;
5 13    controlThread = new Thread(this);
    }

    public void run()
    {
A C E G  for (int i=0; i<count; i++)  a c e
        {
B D F      System.out.print(symbol);  b d
        }
    }

    public void setCount(int count)
    {
7 15    this.count = count;
    }

    public void start()
    {
9 17    controlThread.start();
    }
}
```

In the main thread, the first line to be executed is the first line in the `main` method. Control is then transferred to the default constructor of the `Slasher` class and then to the explicit value constructor. Control is then returned to `main`. Notice that the creation of the `Thread` object does not start a new thread of execution since the `Thread` object is in the "new thread" state.

The next thing that happens in the main thread is that the `Slasher` object's `setCount`

method is called. Notice that this happens in the main thread – only the code in the `run` method (and any code called from it) is executed in the `controlThread` object's thread of execution.

After the return from the `setCount` method, the next thing that happens in the main thread is that the `start()` method in the `slash` object is called. At this point, the `start()` method in the `controlThread` object is called and a second thread of execution is started.

Now, two things are happening "at the same time". In the main thread, control is returned to the `main()` method. In the second thread, the `run()` method of the `slash` object called. Each of these threads then continues as you would expect.

Ultimately, the `start()` method of the `plus` object is called and there are three things happening "at the same time".

S1.5 Race Conditions

Conflicts can arise when more than one thread is using the same object. For example, suppose two threads are both using the same `Dispatcher` object and both are executing the following (modified) `dispatch()` method:

```
public boolean dispatch(String task)
{
   boolean  ok;
   int      vehicle;
   Integer  v;

   ok = false;
   if (availableVehicles.size() > 0)
   {
      v = availableVehicles.removeFirst();
      vehicle = v.intValue();
      sendMessage(vehicle, task);
      ok = true;
   }
   else
   {
      ok = false;
   }


   return ok;
}
```

A very serious problem can arise when there is only one vehicle in the queue.

In particular, suppose the first thread executes all of the code up to and including the evaluation of `(availableVehicles.size() > 0)` and then runs out of time. Further, suppose the seconds thread executes all of the code in this method before it runs out of time. At this point, there will be no vehicles in the queue since the seconds thread removed the only vehicle. Unfortunately, when the first thread starts executing again, it will proceed as if it had never been off the CPU and will attempt to get a vehicle from the queue, causing an exception

This is an example of a *race condition* – code that causes the correctness of a computation to depend on the relative timing of different threads. Specifically, it is an example of a *check-then-act* condition. In between the time that the check was performed and the action was taken the state of the object changed in a way that caused problems.

Though this particular race condition can be avoided using the original implementation of the `dispatch()` method, they are often much more tenacious. Consider, for example, the following method which contains a *read-modify-write* condition:

```
public int getNextIndex()
{
  return ++index;
}
```

The expression `++index` actually performs three operations – load the value, increment the value, and store the value. In other words, it is not an *atomic* operation. As a result, two threads can actually be given the same value for the next index. This happens when the first thread executes the load then stops, the second thread executes the load then stops, the first thread increments then stops, the second thread increments then stops, the first thread stores then stops, and the second thread stores then stops.

S1.6 Synchronization

In theory, race conditions can be prevented in a variety of different ways. Far and away the simplest and the most popular are *synchronized* blocks/methods.[4]

In Java, every object (and class) has a "concurrency protection" object associated with it, called a *monitor* (or an *intrinsic lock*). When a thread of execution reaches a synchronized block or method it attempts to acquire the relevant monitor. (In the case of a synchronized block the relevant object is specified explicitly. In the case of a synchronized method the relevant object is implicitly referenced by `this`.) A thread of execution can only enter a synchronized method/block if it can acquire the relevant monitor. In addition, only one thread can acquire a monitor at a time (making the monitor a *mutex* or *mutual exclusion lock*). A thread of execution that cannot obtain the monitor *blocks*[5] In essence, this means that it does nothing but "periodically" try to obtain the monitor.

---

[4]Classes in the `java.util.concurrent` allow for the use of alternative, and more sophisticated, techniques.

[5]The terms "block" and "blocks" are completely unrelated. "Block" is used as a noun, whereas "blocks" is used as as verb (**not** the plural of "block").

What all of this means is that only one thread at a time can be executing a synchronized block/method, thereby eliminating the possibility of a race condition. When a thread exits a synchronized block/method it releases the monitor, allowing other threads that are blocking to obtain the monitor and proceed.

The *access modifier* `synchronized` is used to indicate that a thread of execution can only enter a method/block if it can obtain the associated monitor. For a method, the `sychronized` modifier must appear where other modifiers (e.g., `public`, `private`, `static`, etc...) can appear, before the return type. For a block, the `synchronized` modifier, followed by parentheses containing the name of the monitor, must appear before the { that defines the start of the block.

A synchronized method is illustrated in the following modified version of the `dispatch()` method in the `Dispatcher` class:

```
public synchronized boolean dispatch(String task)
{
   boolean  ok;
   int      vehicle;
   Integer  v;

   ok = false;
   v  = availableVehicles.removeFirst();

   if (v == null) ok = false;
   else
   {
      vehicle = v.intValue();
      sendMessage(vehicle, task);
      ok = true;
   }

   return ok;
}
```

Since the entire method is synchronized, only one thread can be "in" it a time. Thus, there is no possibility of the threads being interleaved in this method in a way that causes a failure. In essence, the entire method now behaves as if it were atomic.

Unfortunately, an important fault still remains since there are other methods in the class that change state. In particular, both the `dispatch()` method and the `makeVehicleAvailable()` method change the attribute `availableVehicles`. Hence, one thread can be in the `dispatch()` method removing vehicles from the `availableVehicles` queue and another thread can be in the `makeVehicleAvailable()` method adding vehicles to the `availableVehicles` queue. Since the `LinkedList` object `availableVehicles` is not

thread safe, this can cause problems.[6]

Hence, the code in the `makeVehicleAvailable()` method that changes state must also be synchronized, and it must be synchronized with the same monitor. Since the `dispatch()` method implicitly uses `this` as its monitor, the easiest way to do this is to make the `makeVehicleAvailable()` synchronized, as follows:

```
public synchronized void makeVehicleAvailable(int vehicle)
{
   availableVehicles.addLast(new Integer(vehicle));
}
```

It is important to realize that one thread can acquire the same monitor multiple times. That is, monitors in Java are *reentrant*. Java uses a counter to keep track of how many times a thread has acquired a particular monitor.

S1.7 Liveness Failures

A *liveness failure* is a state in which an application/algorithm is unable to make any progress. The most obvious example in single-threaded applications in the "infinite loop". In multi-threaded applications with synchronized blocks/methods, liveness failures can result from much more subtle faults. Two of the most common are *deadlock* and *livelock*.

Deadlock arises when two or more threads are waiting on conditions that can't be satisfied. For example suppose that thread 1 has acquired monitor `m` but needs to acquire monitor `n` to continue, and that thread 2 has acquired monitor `n` but needs to acquire monitor `m` to continue. These two threads are said to be in deadlock (because of a *cyclic locking dependency*).

Livelock arises when a thread can't make progress because it repeatedly attempts an operation that fails. This can occur when two threads are too cooperative (i.e., each attempts to get out of the other's way).

The design of applications that avoid these problems is beyond the scope of this book. Fortunately, if you are careful, they are unlikely to arise in the kinds of applications that are consider here.

S1.8 Performance Failures

In addition to liveness failures, when writing multi-threaded applications one must also be concerned with the possibility of performance failures. That is, it is fairly easy to write multi-threaded applications that do not satisfy performance requirements.

---

[6]The "new" collections in Java, unlike the "original" collections `Hashtable` and `Vector`, are not thread safe by default. It is, however, possible to decorate (in the sense of the Decorator pattern) the "new" collections and make them thread safe. In the case of a `LinkedList` this is done with the static method `synchronizedList()` in the `Collections` class.

In fact, though it's not immediately obvious, the dispatching system implemented thus far is likely to suffer from performance problems. The reason you haven't noticed any performance problems is that, thus far, the messaging process has been simulated using console output. However, notice that the `sendMessage()` method in the `Dispatcher` class is being called from the `dispatch()` method in the `Dispatcher` class which, in turn, is being called by either the `DailyDispatchHandler` or the `RealTimeDispatchHandler`. Hence, in the case of the former, the `sendMessage()` methods is being executed in the `DailyDispatchHandler` object's thread and, in the case of the latter, the `sendMessage()` methods is being executed in the main thread. In both cases, if that thread blocks, the processing of dispatches stops.

A better approach is to have the `dispatch()` method in the `Dispatcher` class return almost immediately, and to have the messaging code execute in another thread. To achieve the first objective it is necessary to add a "task queue" to the `Dispatcher` class and change the `dispatch()` method so that it just adds the task to the task queue and returns. This is implemented below:

```
    public void dispatch(String task)
    {
       // Add a task to the queue
       tasks.add(tasks.size(), task);
    }
```

To achieve the second objective it is necessary to make the `Dispatcher` class implement `Runnable` and add a `Thread` object. This is implemented as follows:

```
import java.util.*;


public class Dispatcher implements Runnable
{
    private          int             numberOfVehicles;
    private          List<Integer>   availableVehicles;
    private          List<String>    tasks;
    private          Thread          dispatchThread;


    public Dispatcher(int n)
    {
       int     i;

       numberOfVehicles = n;
```

```
        availableVehicles =
         Collections.synchronizedList(new LinkedList<Integer>());

        tasks =
         Collections.synchronizedList(new LinkedList<String>());

        for (i=0; i < n; i++) makeVehicleAvailable(i);

        // Start the thread
        dispatchThread = new Thread(this);
        dispatchThread.start();
    }

    private void sendMessage(int vehicle, String message)
    {

        // This method would normally transmit the message
        // to the vehicle.  For simplicity, it now writes it
        // to the screen instead.

        System.out.println(vehicle+"\t"+message+"\n");
        System.out.flush();
    }

}
```

Note that the `LinkedList` objects are made thread safe to avoid any potential race conditions that might arise.

This leads to the following `run()` method:

```
    public void run()
    {
        while (true)
        {
            processPendingDispatches();

            try
            {
                dispatchThread.sleep(1000);
            }
            catch (InterruptedException ie)
            {
                // Shouldn't be interrupted.  If it is,
```

```
            // just continue.
         }
      }
   }
```

and the following `processPendingDispatches()` method:

```
   private void processPendingDispatches()
   {
      int      vehicle;
      Integer  v;
      String   task;

      while ((availableVehicles.size()>0) && tasks.size()>0)
      {
         v = availableVehicles.remove(
                              availableVehicles.size()-1);

         task = tasks.remove(tasks.size()-1);

         vehicle = v.intValue();
         sendMessage(vehicle, task);
      }
   }
```

Note that the `dispatchThread` object is put to sleep for 1000 milliseconds after it processes dispatches so that the other threads can be allowed to modify the two queues.

While this approach works, it is somewhat troubling to be putting the `dispatchThread` object to sleep for an arbitrary amount of time. It would be better for the `dispatchThread` to wait until there are either new tasks to be dispatched or new vehicles to dispatch them to. This can be accomplished this using the `wait()` and `notify()` methods in the `Object class`.

For sake of clarity, this class again uses unsynchronized `LinkedList` objects and, instead, uses an `Object` named `lock` for block-level synchronization as follows:

```
import java.util.LinkedList;


public class Dispatcher implements Runnable
{
    private int                     numberOfVehicles;
    private LinkedList<Integer>     availableVehicles;
```

```
    private LinkedList<String>     tasks;
    private Thread                 dispatchThread;


    private final Object lock = new Object();


    public Dispatcher(int n)
    {
       int     i;

       numberOfVehicles  = n;
       availableVehicles = new LinkedList<Integer>();
       tasks             = new LinkedList<String>();


       for (i=0; i < n; i++)
       {
          makeVehicleAvailable(i);
       }

       dispatchThread = new Thread(this);
       dispatchThread.start();
    }

    private void sendMessage(int vehicle, String message)
    {

       // This method would normally transmit the message
       // to the vehicle.  For simplicity, it now writes it
       // to the screen instead.

       System.out.println(vehicle+"\t"+message+"\n");
       System.out.flush();
    }

}
```

Any code that wants to use the `lock` object for synchronization (i.e., any calls to `wait()`, `notify()`, or `notifyAll()` methods) must appear in a `synchronized` block. This is because a thread must have the `lock` object's monitor before it can call these methods.

So, the `dispatch()` method must now be synchronized with the `lock` attribute:

```
    public void dispatch(String task)
```

```
{
   synchronized(lock)
   {
      // Add a task to the queue
      tasks.addLast(task);

      // Start the processing
      lock.notifyAll();
   }
}
```

as must the `makeVehicleAvailable()` method:

```
public void makeVehicleAvailable(int vehicle)
{
   synchronized(lock)
   {
      // Put the vehicle in the queue
      availableVehicles.addLast(new Integer(vehicle));

      // Start the processing
      lock.notifyAll();
   }
}
```

Now, the `run()` method can "wait" after it processes any pending dispatches:

```
public void run()
{
   while (true)
   {
      synchronized(lock)
      {
         processPendingDispatches();

         try
         {
            lock.wait();
         }
         catch (InterruptedException ie)
```

```
            {
               // Shouldn't be interrupted.  If it is,
               // just continue.
            }
         }
      }
   }
```

The thread is "notified" when there is a new task to dispatch and/or when there a new vehicle is made available.

S1.9 Starting and Stopping Threads

One big problem remains with this last version of the `Dispatcher` – the thread that is handling the dispatches never dies since it never "drops out of" the `run()` method. This problem can be corrected by adding a `boolean` attribute named `keepRunning` and modifying the `run()` method as follows:

```
public void run()
{
   while (keepRunning)
   {
      synchronized(lock)
      {
         processPendingDispatches();

         try
         {
            lock.wait();
         }
         catch (InterruptedException ie)
         {
            // The stop() method was called in
            // another thread
         }
      }
   }

   dispatchThread = null;
}
```

A `stop()` method can then be added as follows:

```
public void stop()
{
   synchronized(lock)
   {
      keepRunning = false;

      // Interrupt the thread in case it
      // is waiting
      dispatchThread.interrupt();
   }
}
```

and even a `start()` method that can be used to start and re-start the `Dispatcher` object:

```
public void start()
{
   if (dispatchThread == null)
   {
      keepRunning    = true;
      dispatchThread = new Thread(this);
      dispatchThread.start();
   }
}
```

Note that the `stop()` method, in addition to assigning `false` to `keepRunning`, calls the `dispatchThread` object's `interrupt()` method. This helps ensure that the thread of execution actually dies. Specifically, the `dispatchThread` object's thread of execution might be waiting in the Not Runnable state when `stop()` is called. If it is never notified to transition to the Runnable state (i.e., if `dispatch()` or `makeVehicleAvailable()` are never called), it will never die.

This class can be tested with the following driver:

```
import java.io.*;
import java.util.Date;

public class Driver
{
    public static void main(String[] args) throws IOException
    {
```

```
        BufferedReader           in;
        DailyDispatchHandler     daily;
        Dispatcher               dispatcher;
        RealTimeDispatchHandler  rt;
        String                   userInput;


        in=new BufferedReader(new InputStreamReader(System.in));

        dispatcher = new Dispatcher(3);


        daily = new DailyDispatchHandler(dispatcher,
                                         "dispatches.txt");
        daily.start();


        // The Dispatcher is started after the
        // DailyDispatchHandler in this example to show
        // that the requests queue up
        try
        {
           Thread.sleep(2000); // Put the current thread to sleep
        }
        catch (InterruptedException ie)
        {
           // Ignore
        }

        dispatcher.start();


        // Start the RealTimeDispatchHandler
        System.out.println("After some dispatches, enter an EOS!");
        rt = new RealTimeDispatchHandler(dispatcher);
        rt.start();


        // Vehicles are put back in the queue
        dispatcher.makeVehicleAvailable(2);
        dispatcher.makeVehicleAvailable(0);

        // Stop the Dispatcher
        dispatcher.stop();
    }
```

```
}
```

S1.10 Volatile Attributes

If you give the discussion above any thought you will quickly realize that the `stop()` method above is going to be executed in a different thread than the `run()` method. Given that this is the case, one must consider whether they are both using the same attributes. If they are, a synchronization problem could arise.

A cursory examination shows that they are, indeed, both using the attribute `keepRunning`. Specifically, the thread that executes the `stop()` method will be "storing" to `keepRunning` and the thread that executes the `run()` method will be "loading" from `keepRunning`.

Now, you might be inclined to think that this is not a problem since the "store" and "load" operations must, surely, be atomic. In fact, there is a problem with this implementation, though it is very subtle. It arises because Java does not ensure that changes to attributes that are made in one thread propogate to other threads in the way you would expect. Specifically, values can be "cached" (e.g., in registers or processor-specific caches) in such a way that they are hidden from other threads.[7] Hence, one has to be concerned about *memory visibility*.

In this case, the problem can be fixed by declaring `keepRunning` to be `volatile` as follows:

```
    private volatile boolean       keepRunning;
```

Volatile attributes are, in essence, attributes that are shared across multiple threads. A "load" of a volatile attribute in any thread always returns the most recent "store" performed in any thread.

REFERENCES AND FURTHER READING

Campione, M., Walrath, K. and Huml, A. (2001) *The Java Tutorial: A Short Course on the Basics* Addison-Wesley Publishing Company, Reading, MA.

Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D. and Lea, D. (2006) *Java Concurrency in Practice* Addison-Wesley Publishing Company, Reading, MA.

Gosling, J., Joy, B., Steele, G. and Bracha, G. (2005) *Java Language Specification* Prentice Hall, Upper Saddle River, NJ.

Herlihy, M. and Shavit, N. (2008) *The Art of Multiprocessor Programming* Morgan Kaufmann, Cambridge, MA.

[7] In addition, the same operations can actually be performed in different orders in different threads. This is done so that the JVM can take full advantage of multiprocessor architectures.

Design Patterns

S2

As defined in Gamma et al. (1995), a *software design pattern* is a "description of communicating objects and classes that is customized to solve a general design problem in a particular context". A design pattern has a lower level of abstraction then an architecture and a higher level of abstraction that an algorithm or data structure. They are generally categorized as either *creational*, *structural*, or *behavioral*.

The book *The Design and Implementation of Multimedia Software* relies on software design patterns quite heavily. The heavy use of design patterns is based on the belief that the study and use of design patterns helps designers avoid the kinds of bad decisions that have been made in the past and benefit from the good ones. Good design patterns can be used "as is" when appropriate, and they can be extended as necessary. The heavy use of design patterns is also based on the belief that design patterns promote better communication (by providing common terminology).

It is interesting (though not central to the main topic of this book) to note that this same basic approach has been used in other design disciplines. For example, Alexander (1977) discusses a pattern language for architectural design and Brown (1986) and Petroski (1994) discuss design paradigms in structural engineering.

S2.1 The Iterator Pattern

The iterator pattern is a behavioral pattern.

S2.1.1 Motivation

One of the most powerful aspects of programming languages is that they give you the ability to perform the same operation on multiple entities using a loop of some kind. Of course, this requires that you have some way of aggregating the entities (e.g., in an array, list, table, etc...).

Unfortunately, traditional looping requires an understanding of the structure of the aggregate object. For example, one loops over the elements in an array somewhat differently from the way one loops over the elements in a vector, and both are very different from the way one loops over a linked data structure.

In the following fragment the aggregate:

```
String    city;

for (int i=0; i < cities.length; i++)
{
   city = (String)cities[i];
   System.out.println(city);
}
```

This fragment contains a `for` loop (which is preferred by many people when using arrays), the termination condition uses the array's `length` attribute, and the elements in the array are accessed using the `[ ]` operator.

In the next fragment the aggregate is an `ArrayList`:

```
String    city;

for (int i=0; i < cities.size(); i++)
{
   city = (String)cities.get(i);
   System.out.println(city);
}
```

Again, this fragment contains a `for` loop, but now the termination condition uses the `size()` method, and the elements in the `Vector` are accessed using the `elementAt()` method.

In the final fragment the aggregate is implemented using a linked data structure:

```
Node      current;
String    city;

current = first;
while (current != null)
{
   city = (String)current.value;
   System.out.println(city);
   current = current.next;
}
```

Now it is more convenient to use a `while` loop that repeatedly assigns `current.next` to `current`, the termination condition involves a `null` pointer, uses the `size()` method, and the elements in the `Node` are accessed using the `.` operator.

Is this really a big problem? Yes, when you realize that an application might loop over the same aggregate object in many different classes and methods. This makes it very difficult to change the aggregate object since it involves changing every loop in a way that can not be done with a global search and replace.

The intent of the iterator pattern is to access the elements of an aggregate object while hiding the internal structure of the aggregate. The `Vector` class and the `Hashtable` class make use of the iterator pattern for just this reason. Both have `elements()` methods that return an `Enumeration`.

S2.1.2 Operations

An iterator must be able to:

1. Reset its "pointer" (or cursor) to the first element.
2. Determine if there are any more elements in the sequence.
3. Move its "pointer" to the next element.
4. Retrieve the "current" element.

This is illustrated in Figure S2.1 on the following page.

S2.1.3 An Example

The following example involves two classes that manage a collection of names. Both include `read()` methods that read the names from a file:

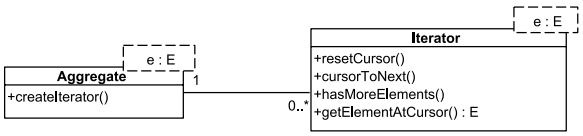

Figure S2.1 The Iterator Pattern

```
    public void read(String fn)
    {
       BufferedReader      in;
       String              line;

       try
       {
          in = new BufferedReader(
             new FileReader(fn));

          while ( (line = in.readLine()) != null)
          {
             names.add(line);
          }

          in.close();
       }
       catch (IOException ioe)
       {
          System.err.println("Problem opening file: "+fn);
          System.exit(1);
       }
    }
```

The `NameList` class uses a `Vector` to manage the names, as follows:

```
import java.io.*;
import java.util.*;

public class NameList
{
    private Vector<String>   names;
```

```
    public NameList()
    {
       names = new Vector<String>();
    }



    public Enumeration<String> elements()
    {
       return names.elements();
    }

}
```

On the other hand, the `NameDatabase` class uses a `Hashtable` for the same purpose:

```
import java.io.*;
import java.util.*;

public class NameDatabase
{
    private Hashtable<String, String>     names;


    public NameDatabase()
    {
       names = new Hashtable<String, String>();
    }


    public void add(String name)
    {
       names.put(name, name);
    }



    public Enumeration<String> elements()
    {
       return names.elements();
    }
```

```
}
```

However, since both use the iterator pattern, an application can use both in the same way (except for their declaration and construction). For example, the following fragment uses the `NameList`:

```
        // To use a NameList
        //
        NameList       names;

        names = new NameList();
```

```
        // Nothing else has to change
        //

        Enumeration<String>    iterator;
        String                 name;


        names.read("people.txt");

        iterator = names.elements();

        while (iterator.hasMoreElements())
        {
            name = iterator.nextElement();
            System.out.println(name);
        }
```

while the following fragment uses the `NameDatabase`:

```
        // To use a NameDatabase
        //
        NameDatabase   names;

        names = new NameDatabase();
```

```
        // Nothing else has to change
        //

        Enumeration<String>    iterator;
        String                 name;


        names.read("people.txt");

        iterator = names.elements();

        while (iterator.hasMoreElements())
        {
            name = iterator.nextElement();
            System.out.println(name);
        }
```

The only difference between the two is in the declaration and instantiation of the aggregate object `names`.

S2.1.4 Other Benefits

It is important to note that, since the `Iterator` keeps track of where it is at any point in time, several objects can be "looping" over the elements in the aggregate at the same time.

For example, the `NamePrinter` class below prints elements in an `Enumeration` in a separate thread of execution:

```
import java.util.*;

public class NamePrinter implements Runnable
{
    private static int   instances = 0;

    private Enumeration  iterator;
    private int          id;
    private Thread       controlThread;



    public NamePrinter(Enumeration names)
    {
       iterator = names;
```

```
        instances++;
        id = instances;

        controlThread = new Thread(this);
        controlThread.start();
    }


    public void run()
    {
        int    delay;
        Random random;
        String name;

        random = new Random(id*System.currentTimeMillis());

        while (iterator.hasMoreElements())
        {
            name = (String)iterator.nextElement();
            System.out.println(id+": "+name);

            try
            {
                delay = random.nextInt(100);
                controlThread.sleep(delay);
            }
            catch (InterruptedException ie)
            {
                // Ignore
            }
        }
    }

}
```

The application below then creates several `NamePrinter` objects for the `Enumeration`. Without any "communication" between them, the `NamePrinter` objects will not print any duplicates.

```
import java.util.*;

public class Driver2
{

    public static void main(String[] args)
    {
```

```
        Enumeration  iterator;
        NameList     names;
        NamePrinter  np1, np2, np3;

        names = new NameList();
        names.read("people.txt");
        iterator = names.elements();

        np1 = new NamePrinter(iterator);
        np2 = new NamePrinter(iterator);
        np3 = new NamePrinter(iterator);

    }

}
```

It is also important to note that the iterator pattern makes it possible to handle a "filtered" list (e.g., names starting with the letter 'A' in exactly the same way that the "unfiltered" version would be handled.

S2.2 The Singleton Pattern

The singleton pattern is a creational pattern.

S2.2.1 Motivation

In some applications, it is important that there be exactly (or no more than) one instance of a particular class. For example, in many windowing systems there is exactly one event queue. As another example, in many word processors there is only one menu bar for all of the documents that are being edited.

Unfortunately, most "traditional" constructors do not provide any assurances about the number of instances that can be created. Hence, the purpose of the singleton pattern is to ensure that a class only has one instance and to provide a point of access to it.

S2.2.2 Operations

A `Singleton` must be able to create an instance of itself if one does not exist, or return the existing instance if it does.

S2.2.3 Implementation

A good way to implement the singleton pattern is illustrated in Figure S2.2 on the next page.

The constructor is made private (or, if you need to allow for generalizations, protected) so that it is not visible. The static variable `exists` is used to keep track of whether the class has been instantiated. If `exists` is `true` then the static variable `instance` contains the existing instance (or a pointer to it). Other classes obtain access to the single instance using the public, static `createInstance()` method. It returns the existing instance or creates one as appropriate.

S2.2.4 An Example

As an example, consider a `FileViewer` that is an encapsulation of a window that displays the contents of a text file. It uses the singleton pattern to stop the proliferation of windows.

The overall structure of the class is as follows (the details of the windowing code are not important for this discussion and, hence, are omitted):

```
public class FileViewer
{

    private static boolean      exists = false;
    private static FileViewer   instance;
```

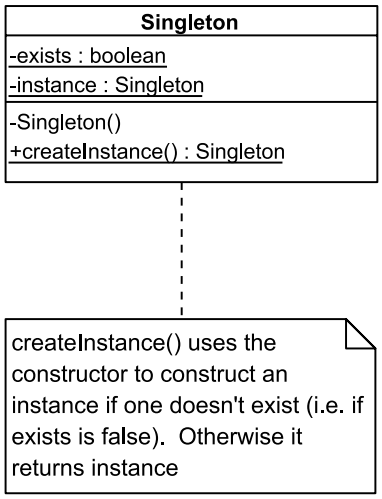

Figure S2.2 The Singleton Pattern

```
    private FileViewer()
    {


        exists = true;
    }

}
```

The `createInstance()` method can then be implemented as follows:

```
    public static FileViewer createInstance()
    {
       if (!exists) instance = new FileViewer();
       return instance;

    }
```

It is worth noting that, in Java, it is not necessary to have the `exists` attribute. Instead, one can check to see whether `instance` is `null`.

The `FileViewer` class can then be used by a `FileChooser` as follows:

```
public void valueChanged(ListSelectionEvent lse)
{
    FileViewer   fv;
    String       fn;

    fn = (String)list.getSelectedValue();

    fv = FileViewer.createInstance();
    fv.load(fn);
}
```

Again, the details of the GUI code do not matter. What's important here is that the `FileChooser` need not be concerned with whether each file will be in it's own `FileViewer` object or not.

S2.2.5 Thread Safety

Since this implementation includes "check then act" logic, problems can arise if the `createInstance()` method might be called by multiple threads. Fortunately, it can be made thread safe relatively easily, if necessary.

One way to make this implementation thread safe is to make the `createInstance()` method synchronized. Another way is to use *eager initialization* (i.e., instantiate the attribute named `instance` when it is declared). This is illustrated in the following fragment:

```
private static FileViewer   instance = new FileViewer();
```

The `createInstance()` method can then be simplified to:

```
public static FileViewer createInstance()
{
    return instance;
}
```

S2.2.6 Other Approaches

Some people recommend that the singleton pattern be implemented with a directory (or registry) of all `Singleton` instances.

Using this approach, each `Singleton` would need to be named, and retrieved from the directory by name. Of course, since you would only want there to be one directory, it would have to be a `Singleton` itself.

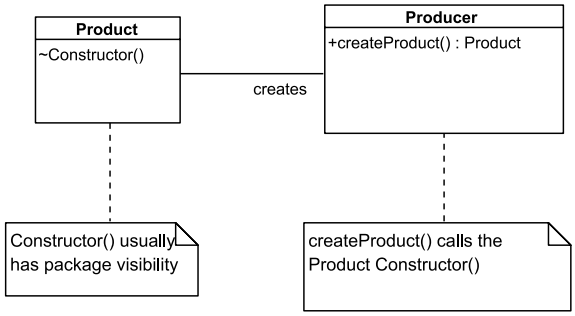

Figure S2.3 The Factory-Method Pattern

S2.3 The Factory-Method Pattern

The factory-method pattern is a creational pattern.

S2.3.1 Motivation

The normal object construction process can be quite limiting. The factory-method pattern attempts to overcome some of this limitations. For example:

- There may be a limit on the number of objects that can be created.
- An object may need to be configured after it is created. For example, consider a set of three objects, each of which needs to contain a reference to the other two.
- A class may need to create an object that will reside on any of several machines.
- A class may need to create a proxy/surrogate for another object and not know it.

S2.3.2 Implementation

A good way to implement the factory-method pattern is illustrated in Figure S2.3.

The `Product` constructor has package/implementation/friend visibility so that it is only visible to the `Producer`. Other classes obtain access to an instance of `Product` using the public `createProduct()` method in the `Producer` class.

The `createProduct()` method may or may not be static. In most cases, only a single `Producer` is needed, in which case the `createProduct()` method and all necessary state variables are mode static. In some cases, their can be multiple `Producer` objects, each of which needs to maintain state information.

S2.3.3 Close Variants

Some `Product` objects are very "expensive" to create (e.g., network connections, complicated look-up tables). In this case, the `Producer` can keep a "pool" of `Product` objects around, and produce them when convenient.

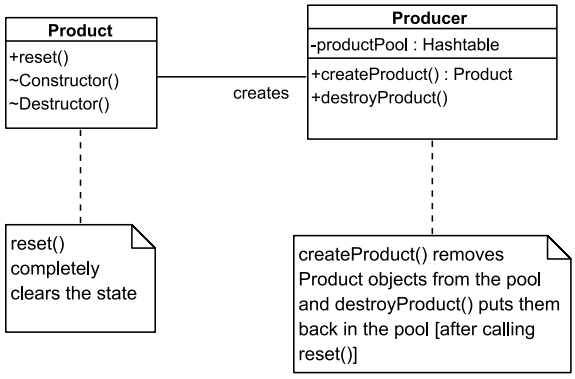

Figure S2.4 The Factory-Method Pattern with an Object Pool

It is also possible to "recycle" `Product` objects (i.e., return them to the "pool"). In this case, the pattern can be implemented as in Figure S2.4.

S2.3.4 An Example

The example that follows makes use of a simple `DirectoryListing` class that provides a method for getting the contents of a directory/folder (that only actually checks the directory/folder if its contents have changed since the last time its contents were requested):

```
import java.io.*;
import java.util.*;

public class DirectoryListing
{
    private File        dir;
    private File[]      files;
    private long        lastTimeCheck;


    DirectoryListing(String path) // package visibility
    {
       dir = new File(path);

       lastTimeCheck = 0;
       update();
    }


    public File[] getContents()
    {
```

```
        update();
        return files;
    }


    private void update()
    {
        long        lastModified;

        lastModified = dir.lastModified();
        if (lastTimeCheck != lastModified)
        {
            lastTimeCheck = lastModified;
            files         = dir.listFiles();
            Arrays.sort(files);
        }
    }
}
```

A `DirectoryListing` object is expensive to create because it accesses the file system.

The `DirectoryListingFactory` uses a "pool" of `DirectoryListing` objects in order to ensure that only one such object exists for any particular path:

```
import java.util.*;

public class DirectoryListingFactory
{
    private Hashtable<String,DirectoryListing>   pool;


    public DirectoryListingFactory()
    {
        pool = new Hashtable<String,DirectoryListing>();
    }


    public DirectoryListing createDirectoryListing(String path)
    {
        DirectoryListing       dl;

        dl = pool.get(path);
        if (dl == null)
        {
            dl = new DirectoryListing(path);
            pool.put(path, dl);
```

```
        }

        return dl;
    }

}
```

The following simple driver illustrates the use of these two classes:

```
import java.io.*;

public class Driver
{

    public static void main(String[] args) throws Exception
    {
        BufferedReader                in;
        DirectoryListing              dir;
        DirectoryListingFactory       factory;
        String                        path;

        factory = new DirectoryListingFactory();

        in = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter a path: ");
        while ((path = in.readLine()) != null)
        {
            dir = factory.createDirectoryListing(path);
            print(dir);
            System.out.print("Enter a path: ");
        }
    }


    public static void print(DirectoryListing dir)
    {
        File[]         files;
        int            i;

        files = dir.getContents();

        for (i=0; i < files.length; i++)
        {
            System.out.println(files[i].getName());
        }
```

```
        System.out.println("\n\n");
    }
}
```

S2.3.5 Related Patterns

The abstract factory pattern should be used when the concrete class of the object being created is not known by the client. For example, the client may need to create a GUI component/widget and may not know the "look-and-feel" of the GUI.

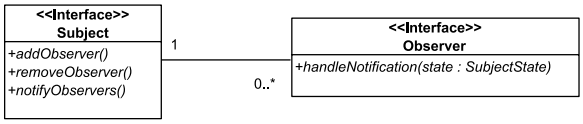

Figure S2.5 The Observer Pattern

S2.4 The Observer Pattern

The observer pattern is a behavioral pattern.

S2.4.1 Motivation

Obviously, objects often need to communicate with each other. Unfortunately, traditional message passing techniques tend to couple objects too tightly in many situations.

For example, consider a spreadsheet application. One might, at any point in time, plot some of the data using a bar chart, a pie chart, or both. How should the data object and the graphical objects communicate? One way is to have the charts periodically poll the data to see if its has changed. This has obvious problems. Another way is to have the data object send a message to the chart objects whenever the data changes. However, the number of chart objects is not known in advance. In addition, each chart object might use information from more than one data object.

As another example, consider a security (e.g., stocks, futures, options) trading application in which there is a `TickReader` that reads "tick by tick" pricing information (e.g., from the Internet) and sends each `Tick` to a `TickWriter` that saves the information in a file and to a `TickerTape` that displays the information on a screen.

One possible design involves thinking of the `TickReader` as actively adding ticks to the `TickWriter` and `TickerTape`. Unfortunately, having the `TickReader` call the `TickWriter` and `TickerTape` reduces the reusability of the `TickReader`. In particular, every `TickReader` must have an associated `TickWriter` and `TickerTape`.

With that in mind, a better approach involves thinking of the `TickWriter` and `TickerTape` as passively listening for "ticks". Then, a `TickReader` need only have a list of (zero or more) `TickListener` objects that it will inform.

The *observer* pattern uses exactly this approach. It does so by defining a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified.

S2.4.2 Structure

The participants in the observer pattern are the `Subject` and the `Observer`. The `Subject` has a list of `Observer` objects, and provides a way for them to add and remove themselves. The `Observer` provides a notification method. This is illustrated Figure S2.5.

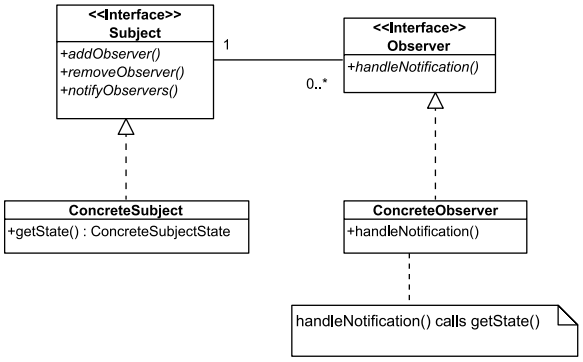

Figure S2.6 Another Version of the Observer Pattern

S2.4.3 Implementation Details

There are at least two different ways for the `Subject` to inform the `Observer` of the specific state changes that have occurred. In the first approach, the state changes are arguments of the `handleNotification()` method. In the second approach, the class that implements the `Subject` interface has a `getState()` method that the class that implements the `Observer` interface uses. This is illustrated in Figure S2.6.

It is also important to note that the `Observer` may have more than one `Subject`. In such situations, the `Observer` needs to be told which `Subject` is doing the notifying. In Java, this is accomplished by having several interfaces that generalize `Observer`. Alternatively, a reference to the `Subject` could be an argument of `handleNotification()`.

S2.4.4 Other Terminology

An `Observer` is sometimes referred to as a "listener", which more accurately portrays the passive nature of the `Observer`. This kind of interaction is also sometimes referred to as *publish-subscribe*. The observer/listener subscribes to the subject, and the subject publishes updates.

S2.4.5 An Example

This section contains an example that illustrates both the observer pattern and important issues related to coupling and cohesiveness. The application considered in this section is a `SillyTextProcessor` that reads lines of text from the console and:

- Counts the number of words that start with an uppercase letter;
- Save the lines to a file; and
- Shows the progress (e.g., the number of lines processed).

One obvious, though bad, design for such an application is to put all of the functionality in a single class as illustrated in Figure S2.7 on the following page.

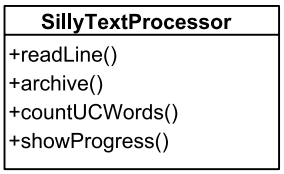

Figure S2.7 A Design for the `SillyTextProcessor` that is Not Cohesive

This design leads to a loop in which all of the lines are read and processed as follows:

```
// Prompt the user
System.out.println("Enter " +maxLines +
                   " lines of text (^Z to end):\n");

// Read from the console
lines = 0;
while ((line = in.readLine()) != null)
{
   // Process each line

}
```

The progress indicator would involve code (inside the loop) like the following:

```
// Indicate the progress
++lines;
bar.setValue(lines);
```

The uppercase word counter would involve code (inside the loop) like the following:

```
// Count the number of words that start with
// uppercase characters (using Java's
// definition of uppercase)
tokenizer = new StringTokenizer(line);
ucCount = 0;
```

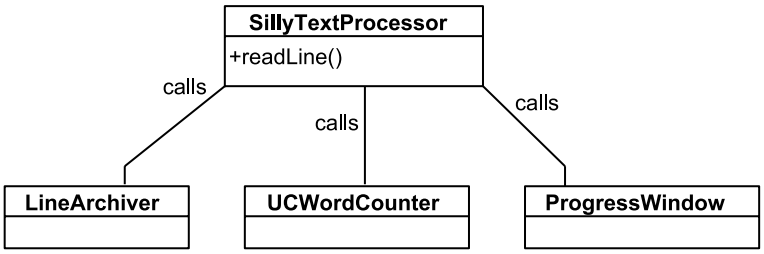

Figure S2.8 A Design of the `SillyTextProcessor` that is Tightly Coupled

```
while (tokenizer.hasMoreTokens())
{
   word = tokenizer.nextToken();
   letter = word.substring(0,1);
   letterUC = letter.toUpperCase();
   if (letter.equals(letterUC)) ++ucCount;
}
System.out.println("Start with uppercase: "+ucCount);
```

Finally, the line archiver would involve code (inside the loop) like the following:

```
// Save the text in a file
out.println(line);
```

This is a classic example of a bad design. The `SillyTextProcessor` class is not *cohesive* and, as a result, is not reusable.

A better approach is to have one class for each of the primary requirements. This leads to `LineReader`, `UCWordCounter`, and `LineArchiver` classes all of which are used by the `SillyTextProcessor` as illustrated in Figure S2.8. The obvious advantage of this design is that each of the classes can, potentially, be used elsewhere.

Without worrying about the details of the "helper" classes, the `SillyTextProcessor` class will now implemented something like the following:

```
// Initialization
in       = new BufferedReader(
               new InputStreamReader(System.in));
bar      = new ProgressWindow(maxLines);
```

```
        archiver = new LineArchiver();


        // Prompt the user
        System.out.println("Enter " +maxLines +
                           " lines of text (^Z to end):\n");

        // Read from the console
        while ((line = in.readLine()) != null)
        {
           // Indicate the progress
           bar.indicateProgress();

           // Count the number of words that start with
           // uppercase characters
           UCWordCounter.count(line);


           // Save the text in a file
           archiver.save(line);
        }
        archiver.close();
```

The problem with this design is that the `SillyTextProcessor` is too *tightly coupled* to the other classes. Specifically, the `SillyTextProcessor` communicates with each of the "helper" classes in a unique way. As a result, there is no easy way to add or remove "helper" classes.

This problem can be corrected using the observer pattern. To do so, one needs to create `LineSubject` and `LineObserver` interfaces, and have the `LineReader` class implement the `LineSubject` interface, and have the `ProgressBar`, `UCWordCounter`, and `LineArchiver` classes implement the `LineObserver` interface. This is illustrated in Figure S2.9 on the following page

The `LineObserver` interface should look something like the following:

```
public interface LineObserver
{

    public void handleLine(LineSubject source);
}
```

and the `LineSubject` interface should look something like the following:

```
public interface LineSubject
{
```

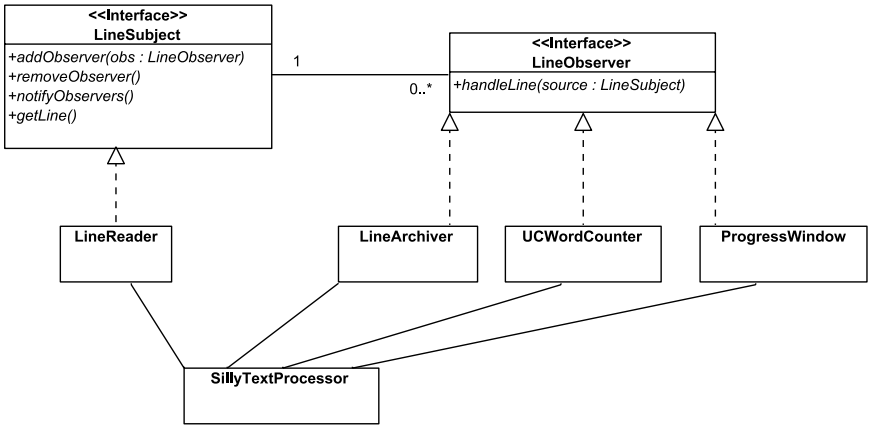

Figure S2.9 A Good Design for the `SillyTextProcessor`

```
    public void addObserver(LineObserver observer);

    public String getLine();


    public void notifyObservers();

    public void removeObserver(LineObserver observer);
}
```

The `LineReader` class must maintain a collection of `LineObserver` objects. This example uses a `List` (in fact, a `LinkedList`) [1] as follows:

```
    private List<LineObserver>                    observers;
```

The `addObserver()` method can then be implemented as follows:

```
    public void addObserver(LineObserver observer)
```

[1] If the collection of observers might be modified in one thread and notified in another then you could, instead, use a `CopyOnWriteArrayList`.

```
    {
       observers.add(observer);
    }
```

and the `notifyObservers()` method can be implemented as follows:

```
    public void notifyObservers()
    {
       Iterator<LineObserver>         i;
       LineObserver                   observer;

       i = observers.iterator();
       while (i.hasNext())
       {
          observer = i.next();
          observer.handleLine(this);
       }
    }
```

It can then use the various `LineObserver` classes in a very "generic" way:

```
    public void start() throws IOException
    {
       // Read from the console and alert listeners
       while ((line = in.readLine()) != null)
       {
          notifyObservers();
       }
    }
```

This reduces the coupling dramatically. The `LineReader` now need only be concerned with "generic" `LineObserver` objects, not with the specifics of the various "helpers".

The `SillyTextProcessor` class now has to construct the subject and each of the observers, and associate the observers with the subject:

```
       // Initialization
       reader   = new LineReader(System.in, maxLines);
       bar      = new ProgressWindow(maxLines);
```

```
        archiver = new LineArchiver();
        counter  = new UCWordCounter();

        reader.addObserver(bar);
        reader.addObserver(archiver);
        reader.addObserver(counter);

        // Prompt the user
        System.out.println("Enter " +maxLines +
                           " lines of text (^Z to end):\n");

        reader.start();
```

The other classes do not change much though they must now implement the `LineObserver` interface.

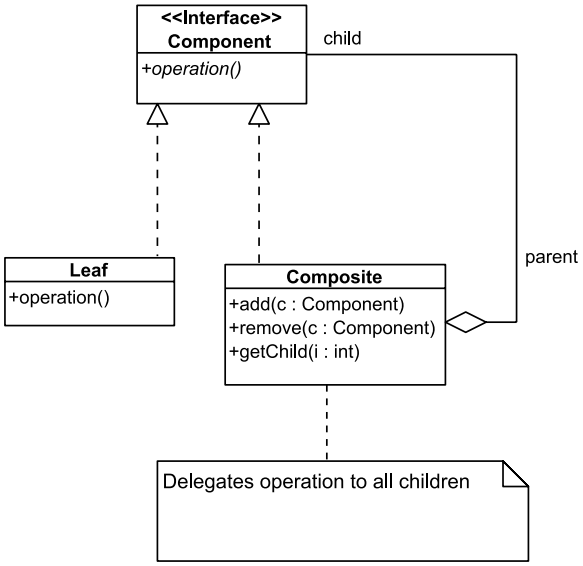

Figure S2.10 The Composite Pattern

S2.5 The Composite Pattern

The composite pattern is a structural pattern.

S2.5.1 Motivation

In many applications, you want to be able to ignore differences between individual objects and compositions of objects. That is, you want to be able to treat all objects (whether composite or not) uniformly.

For example, in many drawing programs you can group several primitive shapes (e.g., lines, rectangles, ovals) together and then perform the same operations on the group as on the primitives (e.g., change the color, re-size) In the "real world", the composite pattern also arises frequently. For example, consider railroads. The components are *locomotives*, *boxcars*, *flatcars*, etc... The containers are *consists* (a group of locomotives), *car sets* (a group of cars), and *trains* (a group of consists, locomotives, and car sets.

S2.5.2 Implementation

The composite pattern involves three class/interfaces, `Component`, `Composite`, and `Leaf`, as illustrated in Figure S2.10. A `Composite` object can contain many `Component` objects, and both the `Leaf` class and the `Composite` class implement the `Component` interface. Thus, a `Composite` can contain both `Leaf` objects and other `Composite` objects.

Each class that implements the `Component` interface must contain a `operation()` method. The `Leaf` class contains a "meaningful" implementation of this method. The `Composite`

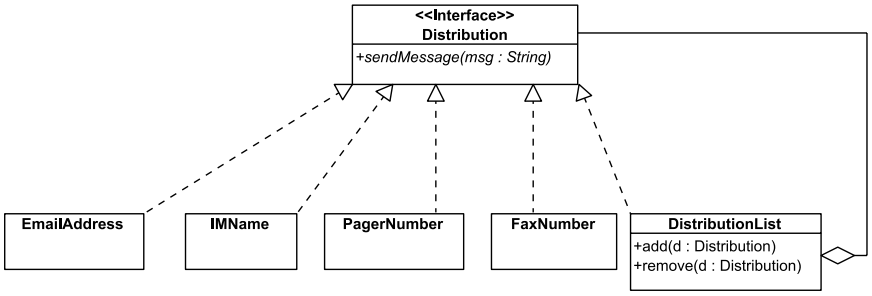

Figure S2.11 An Example of the Composite Pattern

object, on the other hand, simply `operation()` calls to its children. (Note that, though Figure S2.10 on the preceding page only contains on `operation()` method, in practice, there are often multiple "operations" in the `Component` interface.)

S2.5.3 An Example

A good example of the composite pattern is the Message Distribution System illustrated in Figure S2.11. In this example there are several "leaves" – `EmailAddress`, `IMName`, `PagerNumber`, and `FaxNumber`. They all have a `sendMessage` method. A `DistributionList` can contain any and all of these "leaves" as well as other `DistributionList` objects.

The `Distribution` interface is, of course, quite simple:

```
public interface Distribution
{
    public void sendMessage(String msg);
}
```

The following implementation of the `DistributionList` class uses a `Hashtable` to manage its "children":

```
import java.util.*;

public class DistributionList implements Distribution
{
    private Hashtable<Distribution,Distribution>  aggregate;


    public DistributionList()
    {
       aggregate = new Hashtable<Distribution,Distribution>();
```

```
    }


    public void add(Distribution d)
    {
       aggregate.put(d, d);
    }

    public void remove(Distribution d)
    {
       aggregate.remove(d);
    }
}
```

It's `sendMessage()` method delegates to its children using the Iterator pattern (see Section S2.1 on page 30) as follows:

```
    public void sendMessage(String msg)
    {
       Distribution               d;
       Enumeration<Distribution>  iterator;

       // Use the Iterator Pattern to loop through the
       // Distribution objects in the aggregate

       iterator = aggregate.elements();
       while (iterator.hasMoreElements())
       {
          d = iterator.nextElement();

          // Delegate to the Destination
          d.sendMessage(msg);
       }
    }
```

The power of the composite pattern can be illustrated with the following fragment:

```
       DistributionList         cs, math, spam;
       EmailAddress             dean;
```

```
spam = new DistributionList();
dean = new EmailAddress("Tex Avery",
                        "tex@hollywood.edu");

cs = new DistributionList();
cs.add(new EmailAddress("Gilligan",
                        "littlebuddy@island.edu"));
cs.add(new EmailAddress("The Skipper",
                        "skipper@island.edu"));
cs.add(new EmailAddress("Mr. and Mrs. Howell",
                        "magoo@island.edu"));

math = new DistributionList();
math.add(new EmailAddress("Fred Flintstone",
                          "bedrock.edu"));
math.add(new EmailAddress("Barney Rubble",
                          "bedrock.edu"));
math.add(new EmailAddress("Stony Curtis",
                          "hollyrock.edu"));

spam.add(cs);
spam.add(math);
spam.add(dean);

spam.sendMessage("Buy my book!");
```

Note that the `Distribution` list named `spam` contains both an `EmailAddress` object and other `DistributionList` objects.

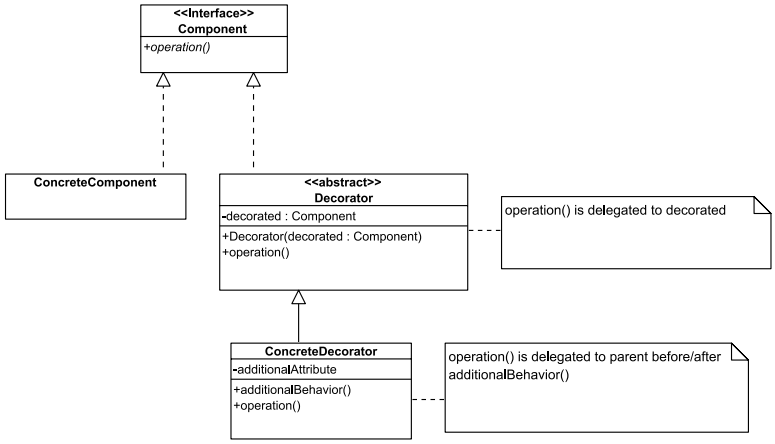

Figure S2.12 The Decorator Pattern

S2.6 The Decorator Pattern

The decorator pattern is a behavioral pattern.

S2.6.1 Motivation

When you specializes an existing class with a class that includes additional methods you are, in effect, adding capabilities to an entire class of objects. There are times, however, when you would like to add capabilities to an indiviual object (at run-time).

For example, consider an application that needs to present output on the console and uses a `Printer` object to do so. At any given time, the application might want to add capabilities to the `Printer` object so that it can, for example, wrap at word boundaries or highlight certain words.

Obviously, one could have `WrappingPrinter` and `HighlightingPrinter` classes that specializes the `Printer` class and provide these capabilitites (and the ability to disable them). However, this approach is fairly inflexible since one can imagine various kinds of capabilties that one might want to add, in various combinations (e.g., highlighting and word-wrapping, just highlighting, just word-wrapping).

The *decorator* pattern uses delegation to add capabilities to individual objects dynamically.

S2.6.2 Structure

The participants in the decorator pattern are the `Component` interface, the `ConcreteComponent` class, the `Decorator` class, and the `ConcreteDecorator` class. The `Component` interface defines the common set of capabilities, and the `ConcreteComponent` class provides those capabilities. The `Decorator` class provides the delegation mechanism and the `ConcreteDecorator` provides the additional capabilities. This is illustrated Figure S2.12.

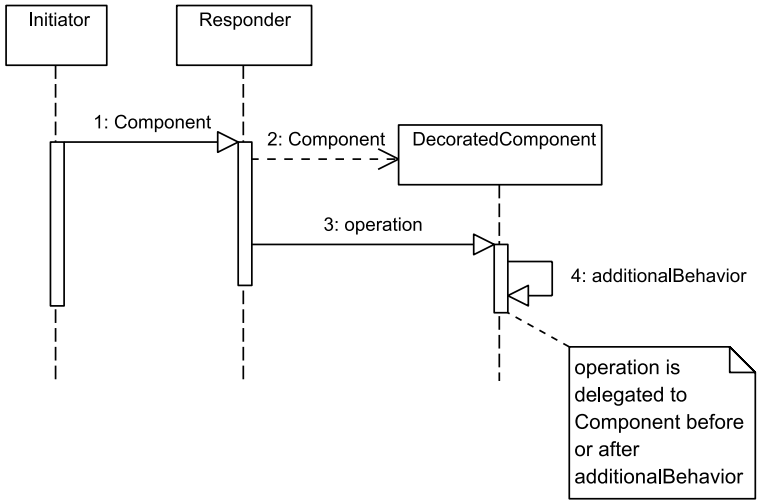

Figure S2.13 A Generic Application of the Decorator Pattern

S2.6.3 A Generic Application

One particularly important type of application of the decorator pattern is illustrated Figure S2.13. In this application, an `Initiator` object passes an object that implements the `Component` interface to a `Responder`. The `Responder` decorates the `Component` (i.e., constructs a `DecoratedComponent` object from the `Component` object) and instructs it to perform `operation()`.

S2.6.4 An Example

Many examples of the decorator pattern involve Graphical User Interface (GUI) widgets. While such examples are instructive, they tend to make people think that the decorator pattern is only used in GUIs, and this is far from the truth.

Hence, this section contains an example that illustrates the use of the decorator pattern in a non-GUI setting. In particular, this section contains an example that is very similar to the `Printer` example discussed above. The overall structure of this example is illustrated Figure S2.14 on the next page.

The `Printer` interface is quite simple:

```
public interface Printer
{
    public abstract void print(String text);
}
```

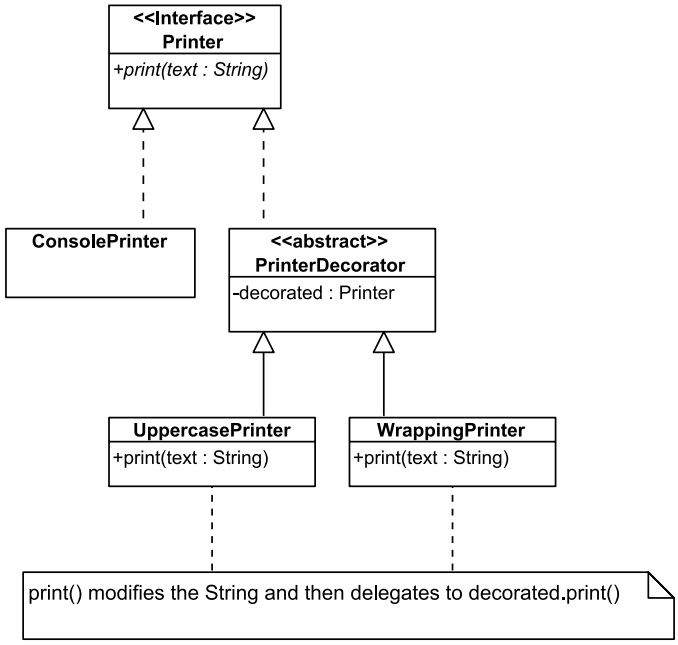

Figure S2.14 An Example of the Decorator Pattern

The `ConsolePrinter` class, as its name implies, simply prints to the console:

```
public class ConsolePrinter implements Printer
{
    public void print(String text)
    {
       System.out.print(text);
    }


}
```

The `PrinterDecorator` class "manages" the `Printer` object for all concrete decorators:

```
public abstract class PrinterDecorator implements Printer
{
    protected Printer     decorated;

    public PrinterDecorator(Printer decorated)
    {
```

```
        this.decorated = decorated;
    }


    public abstract void print(String text);

}
```

The `UppercasePrinter`, which extends `PrinterDecorator`, simply converts the text to uppercase before delegating:

```
public class UppercasePrinter extends PrinterDecorator
{
    public UppercasePrinter(Printer decorated)
    {
       super(decorated);
    }


    public void print(String text)
    {
       decorated.print(text.toUpperCase());
    }
}
```

The `WrappingPrinter`, which also extends `PrinterDecorator`, simply ensures that the text will wrap at word boundaries. It does this by tokenizing the text and delegating on a word-by-word basis:

```
import java.util.*;

public class WrappingPrinter extends PrinterDecorator
{
    protected int         width;


    public WrappingPrinter(Printer decorated, int width)
    {
       super(decorated);
       this.width = width;
    }
```

```
    public void print(String text)
    {
       int                    required, used;
       String                 token;
       StringTokenizer        st;

       st   = new StringTokenizer(text);
       used = 0;

       while (st.hasMoreTokens())
       {
          token    = st.nextToken();
          required = token.length();

          if ((required + used + 1) > width)
          {
             decorated.print("\n");
             decorated.print(token);
             used = required + 1;
          }
          else
          {
             if (used != 0)
             {
                decorated.print(" ");
                ++used;
             }

             decorated.print(token);
             used += required;
          }
       }
    }
}
```

The followin `Driver` class illustrates the flexibility of this design:

```
public class Driver
{
    public static void main(String[] args)
    {
       Printer           printer;
       String            text;
```

```
        text = "This is the text that we will use " +
               "to demonstrate the capabilities "   +
               "of different Printer objects.";


        printer = new ConsolePrinter();
        printer.print(text);

        System.out.print("\n\n");

        printer = new UppercasePrinter(new ConsolePrinter());
        printer.print(text);

        System.out.print("\n\n");

        printer = new WrappingPrinter(new ConsolePrinter(), 20);
        printer.print(text);

        System.out.print("\n\n");

        printer = new WrappingPrinter(
                      new UppercasePrinter(
                          new ConsolePrinter()), 20);
        printer.print(text);
    }

}
```

S2.6.5 Implementation Details

In some situations, one wants to decorate an object that does not implement an explicit interface. In such cases, it is common for the `Decorator` to specialize the class to be decorated.

For example, in Java one might want to decorate a `Graphics2D` so that it uses a coordinate system that has its origing at the lower left (rather than the upper left). One can accomplish this with a `CartesianGraphics` class that extends the `Graphics2D` class (to achieve the necessary "is a" relationship) but delegates to its member `Graphics2D` object. This is illustrated in Figure S2.15 on the following page and is precisely the approached used in Java by various I/O streams.

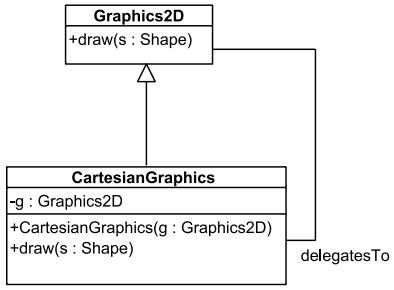

Figure S2.15 Decorating a `Graphics2D` Object in Java

S2.7 The Strategy Pattern

The strategy pattern is a behavioral pattern.

S2.7.1 Motivation

In most situations there are many ways to achieve the same result. In many of these situations, the software designer chooses one (hopefully the best) and implements it. In some situations, one wants to have the flexibility to use more than one.

For example, in numerical optimization, multidimensional search (e.g., the cyclic coordinates method) often involves the repeated application of a one-dimensional search algorithm (e.g., the golden section algorithm, the Fibonacci algorithm). One would like to be able to use different one-dimensional search algorithms at different times.

As another example, in a word processing application, a document formatter might use a paragraph formatter. However, at different times, one might want to use different paragraph formatters (e.g., flush-left-ragged-right, flush-left-and-right, etc...).

The strategy pattern defines a family of interchangeable algorithms for accomplishing the same objective.

S2.7.2 The Pattern

The strategy pattern involves a `Strategy` interface, one or more concrete implementations of that interface, and a `Context` object that uses a `Strategy` to perform a particular task. This is illustrated in Figure S2.16 on the next page.

Either the `Context` object or a third party, constructs the concrete `Strategy` objects and instructs the `Context` object to use one or another. The `Context` object then uses the `operation()` method of the `Strategy` object to perform a particular task.

S2.7.3 An Example

As an example, consider a `Posterizer` that converts a color image to a black and white (not grayscale) image. For each pixel it determines whether the color of that pixel is "closer to"

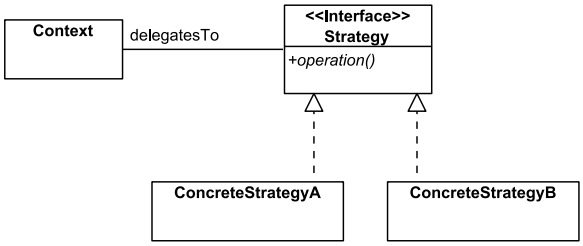

Figure S2.16 The Strategy Pattern

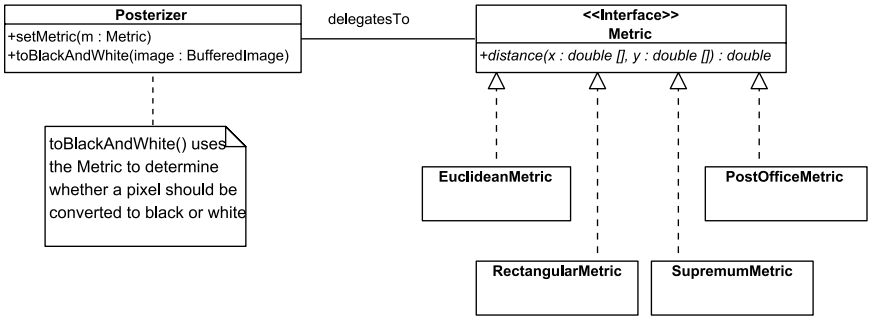

Figure S2.17 An Example of The Strategy Pattern

black or white. If it is "closer to" black, it makes the pixel black. If, on the other hand, it is "closer to" white, it makes the pixel white.

The `Posterizer` needs to determine the distances between the color of the pixel and the colors black and white. However, there are many different ways to calculate the distance between two colors, since colors are represented as 4-dimensional points, one dimension for each of red, green, blue and α. So that a `Posterizer` need not be tied to a particular notion of distance, the strategy pattern is used. The `Metric` interface plays the role of the strategy, and the `EuclideanMetric`, `PostOfficeMetric`, `RectilinearMetric` and `SupremumMetric` classes play the role of the concrete strategies. This is illustrated in Figure S2.17.

The `Metric` interface is straightforward.

```
package math;

public interface Metric
{
    public abstract double distance(double[] x, double[] y);
}
```

This interface is then implemented by the `EuclideanMetric`, `PostOfficeMetric`, `RectilinearMetric` and `SupremumMetric` classes, two of which are included here.

```
package math;

public class      EuclideanMetric
       implements Metric
{
    public double distance(double[] x, double[] y)
    {
       double  result;
       int     n;

       result = 0.0;
       n      = Math.min(x.length, y.length);

       for (int i=0; i<n; i++)
       {
          result += Math.pow(x[i]-y[i], 2.0);
       }

       return Math.sqrt(result);
    }
}
```

```
package math;

public class      SupremumMetric
       implements Metric
{
    public double distance(double[] x, double[] y)
    {
       double  result, term;
       int     n;

       result = Math.abs(x[0]-y[0]);
       n      = Math.min(x.length, y.length);

       for (int i=1; i<n; i++)
       {
          term = Math.abs(x[i]-y[i]);

          if (term > result) result = term;
       }

       return result;
```

```
    }
}
```

The `Posterizer` then uses a particular `Metric` when converting to black and white.

```
import java.awt.*;
import java.awt.image.*;

import math.*;


public class Posterizer
{
    private double[]    x, y;
    private Metric      metric;


    private static final int[] BLACK = {  0,  0,  0};
    private static final int[] WHITE = {255,255,255};

    public Posterizer()
    {
       x = new double[3];
       y = new double[3];
    }


    private double distance(int[] a, int[] b)
    {
       double     result;

       for (int i=0; i<3; i++)
       {
          x[i] = a[i];
          y[i] = b[i];
       }

       result = Double.POSITIVE_INFINITY;
       if (metric != null) result = metric.distance(x, y);

       return result;
    }


    public void setMetric(Metric metric)
```

```
    {
       this.metric = metric;
    }


    public void toBlackAndWhite(BufferedImage image)
    {
       ColorModel  colorModel;
       double      blackDistance, whiteDistance;
       int         height, packedPixel, packedBlack, packedWhite, width;
       int[]       pixel;

       pixel       = new int[3];

       height      = image.getHeight();
       width       = image.getWidth();

       colorModel  = image.getColorModel();

       packedBlack = colorModel.getDataElement(BLACK,0);
       packedWhite = colorModel.getDataElement(WHITE,0);

       for (int x=0; x<width; x++)
       {
          for (int y=0; y<height; y++)
          {
             packedPixel = image.getRGB(x, y);
             colorModel.getComponents(packedPixel, pixel, 0);

             blackDistance = distance(pixel, BLACK);
             whiteDistance = distance(pixel, WHITE);

             if (blackDistance < whiteDistance)
                image.setRGB(x, y, 0x00000000);
             else
                image.setRGB(x, y, 0xFFFFFFFF);
          }
       }
    }

}
```

REFERENCES AND FURTHER READING

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I. and Angel, S. (1977) *A Pattern Language: Towns, Buildings, Construction* Oxford University Press, New York, NY.

Brown, C.B. (1986) "Incomplete Design Paradigms" in *Modelling Human Error in Structural Design and Construction* Nowak, A.S. (ed.), American Society of Civil Engineers, New York, NY.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns:Elements of Reusable Object-Oriented Software* Addison-Wesley Publishing Company, Reading, MA.

Norton, M. (2003) *Composing Software Design Patterns* M.S Thesis, James Madison University.

Petroski, H. (1994) *Design Paradigms* Cambridge University Press, New York, NY.