Java design patterns 101

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

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

Section 1. About this tutorial

Should I take this tutorial?

This tutorial is for Java programmers who want to learn about design patterns as a means of improving their object-oriented design and development skills. After reading this tutorial you will:

* Understand what design patterns are and how they are described and categorized in several well-known catalogs
* Be able to use design patterns as a vocabulary for understanding and discussing object-oriented software design
* Understand a few of the most common design patterns and know when and how they should be used

This tutorial assumes that you are familiar with the Java language and with basic object-oriented concepts such as polymorphism, inheritance, and encapsulation. Some understanding of the Unified Modeling Language (UML) is helpful, but not required; this tutorial will provide an introduction to the basics.

What is this tutorial about?

Design patterns capture the experience of expert software developers, and present common recurring problems, their solutions, and the consequences of those solutions in methodical way.

This tutorial explains:

* Why patterns are useful and important for object-oriented design and development
* How patterns are documented, categorized, and cataloged
* When patterns should be used
* Some important patterns and how they are implemented

Tools

The examples in this tutorial are all written in the Java language. It is possible and sufficient to read the code as a mental exercise, but to try out the code requires a minimal Java development environment. A simple text editor (such as Notepad in Windows or vi in a UNIX environment) and the *Java Development Kit* (version 1.2 or later) are all you need.

A number of tools are also available for creating UML diagrams (see Resources on page 20). These are not necessary for this tutorial.

About the author

David Gallardo is an independent software consultant and author specializing in software internationalization, Java Web applications, and database development. He has been a professional software engineer for over 15 years and has experience with many operating systems, programming languages, and network protocols. David most recently led database and internationalization development at a business-to-business e-commerce company, TradeAccess, Inc. Prior to that, he was a senior engineer in the International Product Development group at Lotus Development Corporation, where he contributed to the development of a cross-platform library providing Unicode and international language support for Lotus products including Notes and 1-2-3.

You can contact David at *david@gallardo.org*.

Section 2. Design patterns overview

A brief history of design patterns

Design patterns were first described by architect Christopher Alexander in his book *A Pattern Language: Towns, Buildings, Construction* (Oxford University Press, 1977). The concept he introduced and called *patterns* -- abstracting solutions to recurring design problems -- caught the attention of researchers in other fields, especially those developing object-oriented software in the mid-to-late 1980s.

Research into software design patterns led to what is probably the most influential book on object-oriented design: *Design Patterns: Elements of Reusable Object-Oriented Software* , by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995; see Resources on page 20). These authors are often referred to as the "Gang of Four" and the book is referred to as the Gang of Four (or GoF) book.

The largest part of *Design Patterns* is a catalog describing 23 design patterns. Other, more recent catalogs extend this repertoire and most importantly, extend coverage to more specialized types of problems. Mark Grand, in *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*, adds patterns addressing problems involving concurrency, for example, and *Core J2EE Patterns: Best Practices and Design Strategies* by Deepak Alur, John Crupi, and Dan Malks focuses on patterns for multi-tier applications using Java 2 enterprise technologies.

There is an active pattern community that collects new patterns, continues research, and takes leads in spreading the word on patterns. In particular, the Hillside Group sponsors many conferences including one introducing newcomers to patterns under the guidance of experts. Resources on page 20 provides additional sources of information about patterns and the pattern community.

Pieces of a pattern

The Gang of Four described patterns as "a solution to a problem in a context". These three things -- problem, solution, and context -- are the essence of a pattern. For documenting the pattern it is additionally useful to give the pattern a name, to consider the consequences using the pattern will have, and to provide an example or examples.

Different catalogers use different templates to document their patterns. Different catalogers also use different names for the different parts of the pattern. Each catalog also varies somewhat in the level of detail and analysis devoted to each pattern. The next several panels describe the templates used in *Design Patterns* and in *Patterns in Java*.

Design Patterns template

Design Patterns uses the following template:

* Pattern name and classification: A conceptual handle and category for the pattern
* Intent: What problem does the pattern address?

* Also known as: Other common names for the pattern
* Motivation: A scenario that illustrates the problem
* Applicability: In what situations can the pattern be used?
* Structure: Diagram using the Object Modeling Technique (OMT)
* Participants: Classes and objects in design
* Collaborations: How classes and objects in the design collaborate
* Consequences: What objectives does the pattern achieve? What are the tradeoffs?
* Implementation: Implementation details to consider, language-specific issues
* Sample code: Sample code in Smalltalk and C++
* Known uses: Examples from the real world
* Related patterns: Comparison and discussion of related patterns

Patterns in Java template

Patterns in Java uses the following template:

* Pattern Name: The name and a reference to where it was first described
* Synopsis: A very short description of the pattern
* Context: A description of the problem the pattern is intended to solve
* Forces: A description of the considerations that lead to the solution
* Solution: A description of the general solution
* Consequences: Implications of using the pattern
* Implementation: Implementation details to consider
* Java API Usage: When available, an example from the Java API is mentioned
* Code example: A code example in the Java language
* Related patterns: A list of related patterns

Learning patterns

The most important things to learn at first is the intent and context of each pattern: what problem, and under what conditions, the pattern is intended to solve. This tutorial covers some of the most important patterns, but skimming through a few catalogs and picking out this information about each pattern is the recommended next step for the diligent developer. In *Design Patterns,* the relevant sections to read are "Intent," "Motivation," and Applicability." In *Patterns in Java,* the relevant sections are "Synopsis," "Context," and "Forces and Solution."

Doing the background research can help you identify a pattern that lends itself as a solution to a design problem you're facing. You can then evaluate the candidate pattern more closely for applicability, taking into account the solution and its consequences in detail. If this fails, you can look to related patterns.

In some cases, you might find more than one pattern that can be used effectively. In other cases, there may not be an applicable pattern, or the cost of using an applicable pattern, in terms of performance or complexity, may be too high, and an *ad hoc* solution may be the best way to go. (Perhaps this solution can lead to a new pattern that has not yet been documented!)

Using patterns to gain experience

A critical step in designing object-oriented software is discovering the objects. There are various techniques that help: use cases, collaboration diagrams, or Class-Responsibility-Collaboration (CRC) cards, for example -- but discovering the objects is the hardest step for inexperienced designers to get right.

Lack of experience or guidance can lead to too many objects with too many interactions and therefore dependencies, creating a monolithic system that is hard to maintain and impossible to reuse. This defeats the aim of object-oriented design.

Design patterns help overcome this problem because they teach the lessons distilled from experience by experts: patterns document expertise. Further, patterns not only describe how software is structured, but more importantly, they also describe how classes and objects interact, especially at run time. Taking these interactions and their consequences explicitly into account leads to more flexible and reusable software.

When not to use patterns

While using a pattern properly results in reusable code, the consequences often include some costs as well as benefits. Reusability is often obtained by introducing encapsulation, or indirection, which can decrease performance and increase complexity.

For example, you can use the Facade pattern to wrap loosely related classes with a single class to create a single set of functionality that is easy to use. One possible application might be to create a facade for the Java Internationalization API. This approach might be reasonable for a stand-alone application, where the need to obtain text from resource bundles, format dates and time, and so on, is scattered in various parts of the applications. But this may not be so reasonable for a multitier enterprise application that separates

presentation logic from business. If all calls to the Internationalization API are isolated in a presentation module -- perhaps by wrapping them as JSP custom tags -- it would be redundant to add yet another layer of indirection.

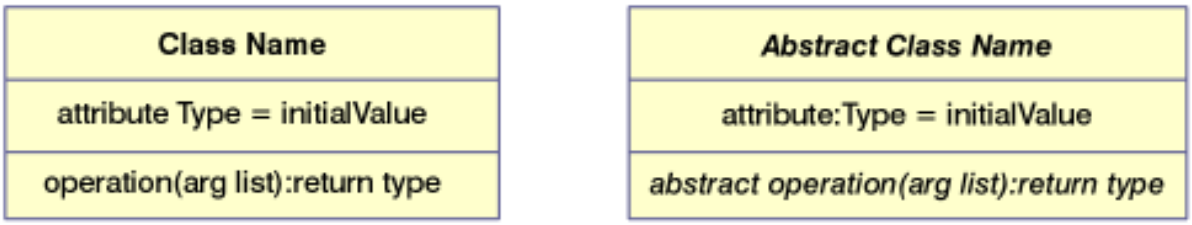
Another example of when patterns should be used with care is discussed in Concurrency patterns on page 18 , regarding the consequences of the Single Thread Execution pattern.

As a system matures, as you gain experience, or flaws in the software come to light, it's good to occasionally reconsider choices you've made previously. You may have to rewrite ad hoc code so that it uses a pattern instead, or change from one pattern to another, or remove a pattern entirely to eliminate a layer of indirection. Embrace change (or at least prepare for it) because it's inevitable.

Section 3. A brief introduction to UML class diagrams

Class diagrams

UML has become the standard diagramming tool for object-oriented design. Of the various types of diagrams defined by UML, this tutorial only uses class diagrams. In class diagrams, classes are depicted as boxes with three compartments:



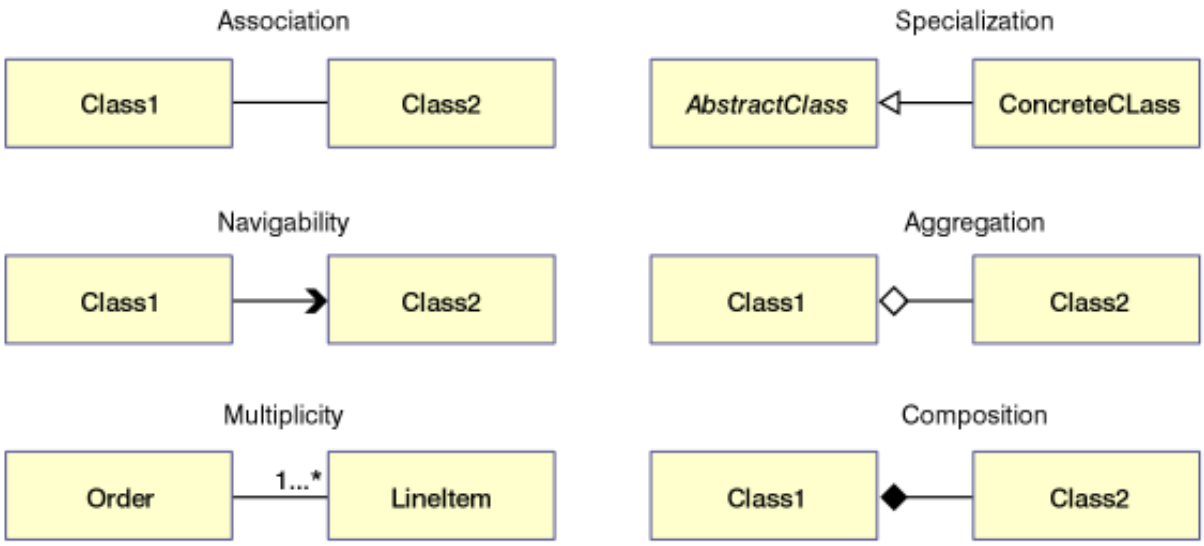
The top compartment contains the class name; if the class is abstract the name is italicized. The middle compartment contains the class attributes (also called *properties*, or variables). The bottom compartment contains the class methods (also called *operations*). Like the class name, if a method is abstract, its name is italicized.

Depending on the level of detail desired, it is possible to omit the properties and show only the class name and its methods, or to omit both the properties and methods and show only the class name. This approach is common when the overall conceptual relationship is being illustrated.

Associations between classes

Any interaction between classes is depicted by a line drawn between the classes. A simple line indicates an association, usually a conceptual association of any unspecified type. The line can be modified to provide more specific information about the association. *Navigability* is indicated by adding an open arrowhead. *Specialization*, or subclassing, is indicated by adding a triangular arrowhead. Cardinal numbers (or an asterisk for an unspecified plurality) can also be added to each end to indicate relationships, such as one-to-one and many-to-one.

The following diagrams show these different types of associations:



Resources on page 20 provides further reading on UML and Java language associations.

Section 4. Creational patterns

Overview

Creational patterns prescribe the way that objects are created. These patterns are used when a decision must be made at the time a class is instantiated. Typically, the details of the classes that are instantiated -- what exactly they are, how, and when they are created -- are encapsulated by an abstract superclass and hidden from the client class, which knows only about the abstract class or the interface it implements. The specific type of the concrete class is typically unknown to the client class.

The Singleton pattern, for example, is used to encapsulate the creation of an object in order to maintain control over it. This not only ensures that only one is created, but also allows lazy instantiation; that is, the instantiation of the object can be delayed until it is actually needed. This is especially beneficial if the constructor needs to perform a costly operation, such as accessing a remote database.

The Singleton pattern

This code demonstrates how the Singleton pattern can be used to create a counter to provide unique sequential numbers, such as might be required for use as primary keys in a database:

```
// Sequence.java
 public class Sequence {
   private static Sequence instance;
   private static int counter;
   private Sequence()
   {
     counter = 0; // May be necessary to obtain
                  // starting value elsewhere...
   }
   public static synchronized Sequence getInstance()
   {
     if(instance==null) // Lazy instantiation
     {
       instance = new Sequence();
     }
     return instance;
   }
   public static synchronized int getNext()
   {
     return ++counter;
   }
 }
```

Some things to note about this implementation:

* **Synchronized** methods are used to ensure that the class is thread-safe.
* This class cannot be subclassed because the constructor is **private**. This may or may not be a good thing depending on the resource being protected. To allow subclassing, the visibility of the constructor should be changed to **protected**.

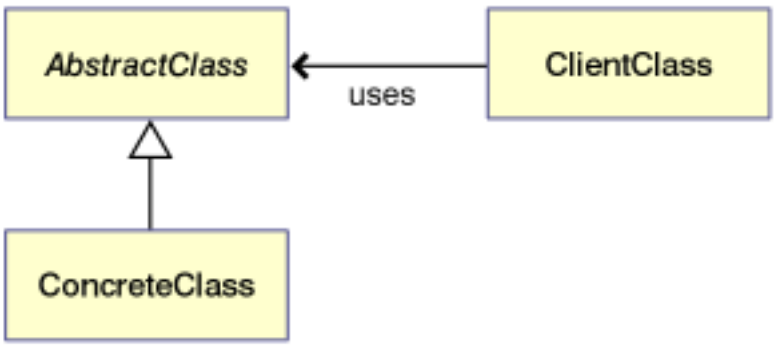
* Object serialization can cause problems; if a Singleton is serialized and then deserialized more than once, there will be multiple objects and not a singleton.

The Factory Method pattern

In addition to the Singleton pattern, another common example of a creational pattern is the Factory Method. This pattern is used when it must be decided at run time which one of several compatible classes is to be instantiated. This pattern is used throughout the Java API. For example, the abstract **`Collator`** class's **`getInstance()`** method returns a collation object that is appropriate for the default locale, as determined by **`java.util.Locale.getDefault()`**:

```
Collator defaultCollator = getInstance();
```

The concrete class that is returned is actually always a subclass of **`Collator`**, **`RuleBasedCollator`**, but that is an unimportant implementation detail. The interface defined by the abstract **`Collator`** class is all that is required to use it.



Section 5. Structural patterns

Overview

Structural patterns prescribe the organization of classes and objects. These patterns are concerned with how classes inherit from each other or how they are composed from other classes.

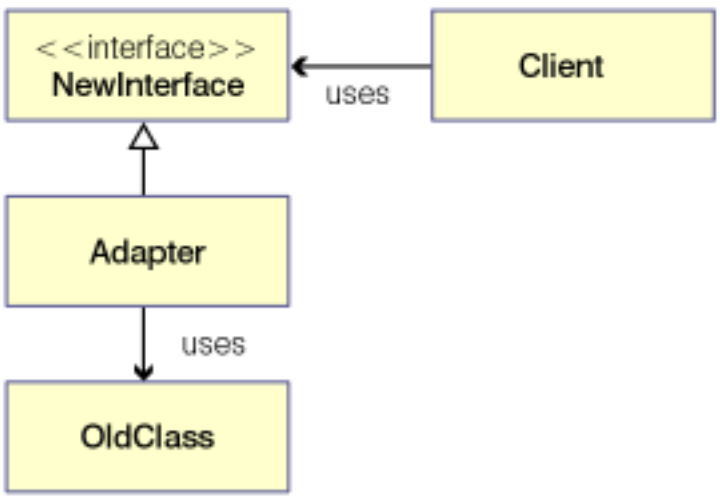
Common structural patterns include Adapter, Proxy, and Decorator patterns. These patterns are similar in that they introduce a level of indirection between a client class and a class it wants to use. Their intents are different, however. Adapter uses indirection to modify the interface of a class to make it easier for a client class to use it. Decorator uses indirection to add behavior to a class, without unduly affecting the client class. Proxy uses indirection to transparently provide a stand-in for another class.

The Adapter pattern

The Adapter pattern is typically used to allow the reuse of a class that is similar, but not the same, as the class the client class would like to see. Typically the original class is capable of supporting the behavior the client class needs, but does not have the interface the client class expects, and it is not possible or practical to alter the original class. Perhaps the source code is not available, or it is used elsewhere and changing the interface is inappropriate.

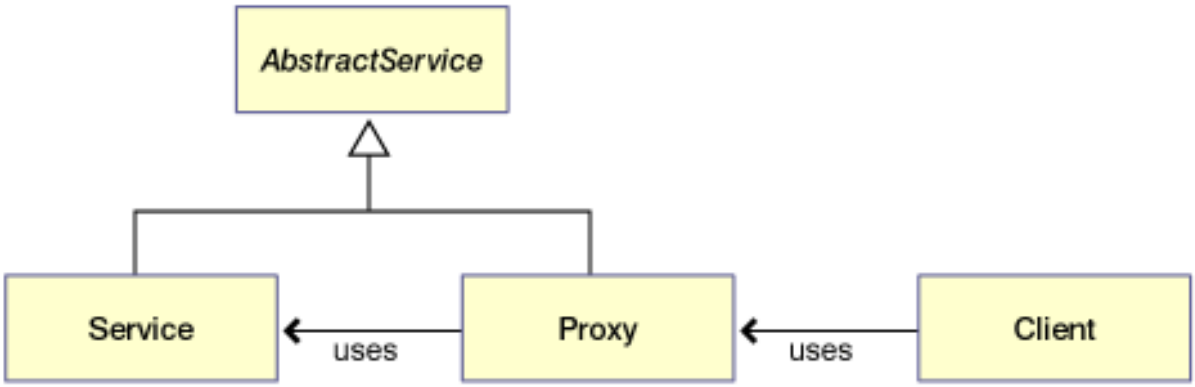
Here is an example that wraps **`OldClass`** so a client class can call it using a method, **`NewMethod()`** defined in **`NewInterface`**:

```
public class OldClassAdapter implements NewInterface {
  private OldClass ref;
  public OldClassAdapter(OldClass oc)
  {
    ref = oc;
  }
    public void NewMethod()
  {
     ref.OldMethod();
  }
}
```



The Proxy and Decorator patterns

A Proxy is a direct stand-in for another class, and it typically has the same interface as that class because it implements a common interface or an abstract class. The client object is not aware that it is using a proxy. A Proxy is used when access to the class the client would like to use must be mediated in a way that is apparent to the client -- because it requires restricted access or is a remote process, for example.



Decorator, like Proxy, is also a stand-in for another class, and it also has the same interface as that class, usually because it is a subclass. The intent is different, however. The purpose of the Decorator pattern is to extend the functionality of the original class in a way that is transparent to the client class.

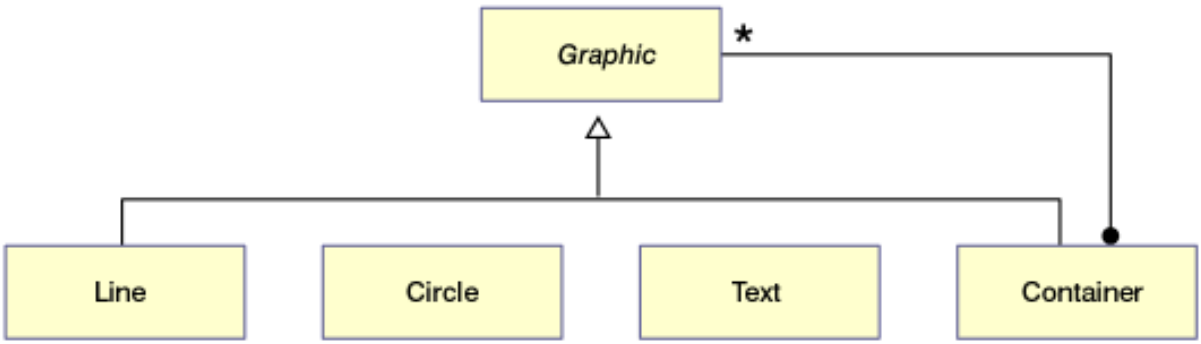
Examples of the Decorator pattern in the Java API are found in the classes for processing input and output streams. **`BufferedReader()`**, for example, makes reading text from a file convenient and efficient:

```
BufferedReader in = new BufferedReader(new FileReader("file.txt"));
```

The Composite pattern

The Composite pattern prescribes recursive composition for complex objects. The intent is to allow all component objects to be treated in a consistent manner. All objects, simple and complex, that participate in this pattern derive from a common abstract component class that defines common behavior.

Forcing relationships into a part-whole hierarchy in this way minimizes the types of objects that our system (or client subsystem) needs to manage. A client of a paint program, for example, could ask a line to draw itself in the same way it would ask any other object, including a composite object.


Graphic
*
Line
Circle
Text
Container

Section 6. Behavioral patterns

Overview

Behavioral patterns prescribe the way objects interact with each other. They help make complex behavior manageable by specifying the responsibilities of objects and the ways they communicate with each other.

The Observer pattern

Observer is a very common pattern. You typically use this pattern when you're implementing an application with a Model/View/Controller architecture. The Model/View part of this design is intended to decouple the presentation of data from the data itself.

Consider, for example, a case where data is kept in a database and can be displayed in multiple formats, as a table or a graph. The Observer pattern suggests that the display classes register themselves with the class responsible for maintaining the data, so they can be notified when the data changes, and so they can update their displays.

The Java API uses this pattern in the event model of its AWT/Swing classes. It also provides direct support so this pattern can be implemented for other purposes.

The Java API provides an **`Observable`** class that can be subclassed by objects that want to be observed. Among the methods **`Observable`** provides are:

* **`addObserver(Observer o)`** is called by **`Observable`** objects to register themselves.
* **`setChanged()`** marks the **`Observable`** object as having changed.
* **`hasChanged()`** tests if the **`Observable`** object has changed.
* **`notifyObservers()`** notifies all observers if the **`Observable`** object has changed, according to **`hasChanged()`**.

To go along with this, an **`Observer`** interface is provided, containing a single method that is called by an **`Observable`** object when it changes (providing the **`Observer`** has registered itself with the **`Observable`** class, of course):

```
public void update(Observable o, Object arg)
```

The following example demonstrates how an Observer pattern can be used to notify a display class for a sensor such as temperature has detected a change:

```
import java.util.*;
class Sensor extends Observable {
private int temp = 68;
  void takeReading()
  {
   double d;
```

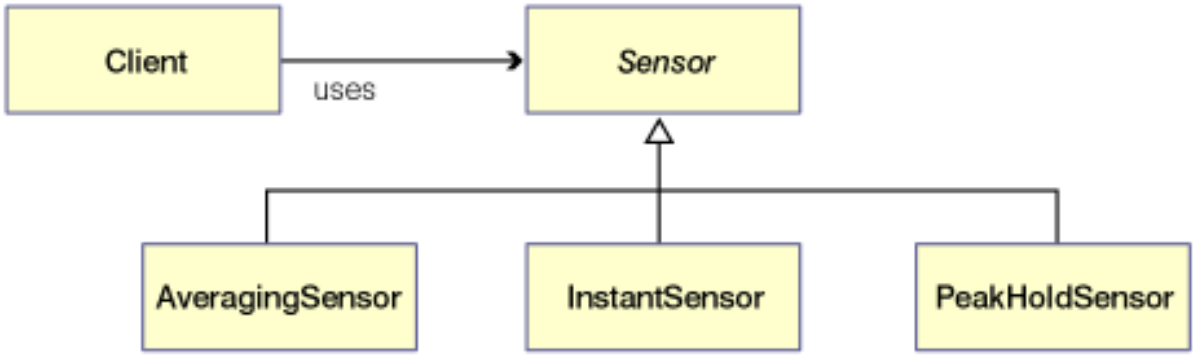
```
      d =Math.random();
      if(d>0.75)
      {
        temp++;
        setChanged();
      }
      else if (d<0.25)
      {
        temp--;
        setChanged();
      }
      System.out.print("[Temp: " + temp + "]");
   }
   public int getReading()
   {
     return temp;
   }
}
public class Display implements Observer {
   public void update(Observable o, Object arg)
   {
      System.out.print("New Temp: " +  ((Sensor) o).getReading());
   }
   public static void main(String []ac)
   {
      Sensor sensor = new Sensor();
      Display display = new Display();
      // register observer with observable class
      sensor.addObserver(display);
      // Simulate measuring temp over time
      for(int i=0; i < 20; i++)
      {
         sensor.takeReading();
         sensor.notifyObservers();
         System.out.println();
      }
   }
}
```

The Strategy and Template patterns

Strategy and Template patterns are similar in that they allow different implementations for a fixed set of behaviors. Their intents are different, however.

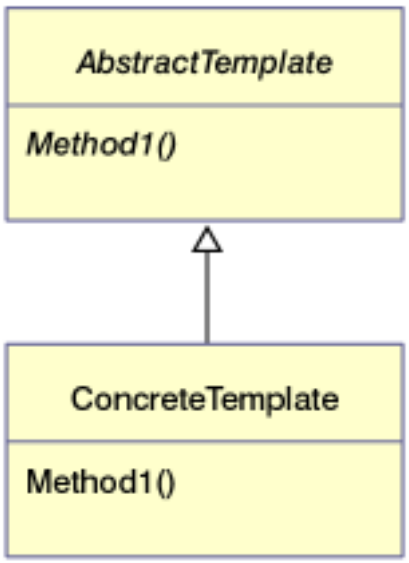
Strategy is used to allow different implementations of an algorithm, or operation, to be selected dynamically at run time. Typically, any common behavior is implemented in an abstract class and concrete subclasses provide the behavior that differs. The client is generally aware of the different strategies that are available and can choose between them.

For example, an abstract class, **`Sensor`**, could define taking measurements and concrete subclasses would be required to implement different techniques: one might provide a running average, another might provide an instantaneous measurement, and yet another might hold a peak (or low) value for some period of time.



The intention of the Template pattern is not to allow behavior to be implemented in different ways, as in Strategy, but rather to ensure that certain behaviors are implemented. In other words, where the focus of Strategy is to allow variety, the focus of Template is to enforce consistency.

The Template pattern is implemented as an abstract class and it is often used to provide a blueprint or an outline for concrete subclasses. Sometimes this is used to implement hooks in a system, such as an application framework.



Section 7. Concurrency patterns

Overview

Concurrency patterns prescribe the way access to shared resources is coordinated or sequenced. By far the most common concurrency pattern is Single Thread Execution, where it must be ensured that only one thread has access to a section of code at a time. This section of code is called a *critical section*, and typically it is a section of code that either obtains access to a resource that must be shared, such as opening a port, or is a sequence of operations that should be atomic, such as obtaining a value, performing calculations, and then updating the value.

The Single Thread Execution pattern

The Singleton pattern we discussed earlier contains two good examples of the Single Thread Execution pattern. The problem motivating this pattern first arises because this example uses lazy instantiation -- delaying instantiating until necessary -- thereby creating the possibility that two different threads may call **`getInstance()`** at the same time:

```
public static synchronized Sequence getInstance()
{
  if(instance==null) // Lazy instantiation
  {
    instance = new Sequence();
  }
  return instance;
}
```

If this method were not protected against simultaneous access with **`synchronized`**, each thread might enter the method, test and find that the static instance reference is null, and each might try to create a new instance. The last thread to finish wins, overwriting the first thread's reference. In this particular example, that might not be so bad -- it only creates an orphaned object that garbage collector will eventually clean up -- but had there been a shared resource that enforced single access, such as opening a port or opening a log file for read/write access, the second thread's attempt to create an instance would have failed because the first thread would have already obtained exclusive access to the shared resource.

Another critical section of code in the Singleton example is the **`getNext()`** method:

```
public static synchronized int getNext()
{
  return ++counter;
}
```

If this is not protected with **`synchronized`**, two threads calling it at the same time might obtain the same current value and not the unique values this class is intended to provide. If this were being used to obtain primary keys for a database insert, the second attempt to insert with same primary key would fail.

As we discussed earlier, you should always consider the cost of using a pattern. Using

synchronized works by locking the section of code when it is entered by one thread and blocking any other threads until the first thread is finished. If this is code used frequently by many threads, this could cause a serious degradation in performance.

Another danger is that two threads could become deadlocked if one thread is blocked at one critical section waiting for the second, while the second thread is blocked at another critical section, waiting for the first.

Section 8. Wrapup

Summary

Design patterns are a valuable tool for object-oriented design for a number of important reasons:

* Patterns provide "...a solution to a problem in a context." (*Design Patterns*, Gamma, Helm, Johnson, and Vlissides).
* Patterns capture the expertise of experienced designers in a methodical way and make them available as design tools and learning tool for non-experts.
* Patterns provide a vocabulary for discussing object-oriented design at a significant level of abstraction.
* Patterns catalogs serve as a glossary of idioms that help in understanding common, but complex solutions to design problems.

Resources

Books

* *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995) is probably the most influential resource on object-oriented design. Chapters 1, 2, and 6 are essential reading for understanding object-oriented design in general or, in particular, the role of patterns in object-oriented design.
* *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML* by Mark Grand (Wiley, 1998) is not as well written as *Design Patterns*, especially regarding general object-oriented design issues, but the patterns in the catalog are easier to understand, particularly because the examples are written using the Java language and the recommendations address issues common for Java developers.
* *Core J2EE Patterns: Best Practices and Design Strategies* by Deepak Alur, John Crupi, and Dan Malks (Prentice Hall, 2001) is a catalog of patterns for the design and architecture of multitier enterprise applications.
* *UML Distilled: Applying the Standard Object Modeling Language* by Martin Fowler with Kendall Scott (Addison-Wesley, 2000) offers an excellent introduction to the essentials of UML. It includes a short but valuable discussion of using Class-Responsibility-Collaboration cards for object-oriented design.
* *The Unified Modeling Language User Guide* by Grady Booch, Ivar Jacobson, and James Rumbaugh (Addison-Wesley, 1998) is helpful when you need more than just the essentials.

Web resources

* developerWorks has two good introductory articles on the Java programming language and on object-oriented design in general:
    * *"The OO design process: Getting started"* by Allen Holub
    * *"The object primer: Using object-oriented techniques to develop software"* by Scott W. Ambler
* There are also several articles on the Java language, patterns, and UML:
    * "A UML workbook" (*Part 1*, *Part 2*, and *Part 3*) by Granville Miller
    * *"An overview of object relationships: The basics of UML and Java associations"* by Scott W. Ambler
    * *"Use your singletons wisely: Know when to use singletons, and when to leave them behind"* by J. B. Rainsberger
    * *"Developing Java solutions using Design Patterns"* by Kelby Zordrager
* There are several Web sites with good information on patterns. The Hillside Group plays a major role in the pattern community and its site, in particular, is an excellent starting point:
    * *The Hillside Group Patterns home page*
    * *Portland Pattern Repository*
    * *Brad Appleton's Software Patterns Links*
* *"Christopher Alexander: An Introduction for Object-Oriented Designers"* is an interesting read for those wanting to learn more about the father of design patterns.
* The most discussed and dissected pattern, in this tutorial and elsewhere, is the Singleton pattern. Here are two articles that cover the topic thoroughly from different perspectives:
    * *"Implementing the Singleton Pattern in Java"* by Rod Waldhoff
    * *"When is a singleton not a singleton?"* by Joshua Fox

UML tools

* UML tools are of two types: those that provide CASE (Computer Aided Software Engineering) and those that are only for creating diagrams. The CASE tools use UML diagrams to generate code. They can also reverse-engineer code and generate diagrams. Some people find these capabilities to be useful. Rational Rose and Together/J are in this category, while ArgoUML and SmartDraw provide only drawing capabilities:
    * *Rational Rose* is an industrial-strength software design tool (and much more) that

strictly enforces its interpretation of UML. This can be frustrating if developers are using it for conceptual or informal designs. It's expensive, but you can download a 15-day evaluation version to test out.

* *Together/J* is also an industrial-strength software design tool that enforces its interpretation of UML. Download the free whiteboard version (which doesn't have code-generation or round-trip engineering) for a test ride.

* *ArgoUML* is a free, Java language, open source, UML diagramming tool. It's much smaller than Rational Rose or Together/J because it doesn't have the code generation and other engineering tools, and it is significantly easier to use. Be aware that the current version, 0.9.5 (as of December 2001), is beta and has significant bugs.

* *SmartDraw* is an inexpensive general-purpose diagramming tool that includes support for UML. Because it is designed as a general-purpose tool, it can be awkward to use for UML. Don't miss the comprehensive *UML tutorial* from SmartDraw.

Feedback

Please send us your feedback on this tutorial. We look forward to hearing from you! Additionally, you are welcome to contact the author, David Gallardo, directly at *david@gallardo.org*.

Colophon

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

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