Design Patterns and CodePro [1, 2]

by Raphael Enns

Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada

Last revised: March 10, 2004

Overview:

In this tutorial, we will explain what design patterns are and how they are useful. This tutorial will describe several design patterns in detail, giving a simple example in Java for each. We will also look at a plug-in for Eclipse that helps create design patterns.

About Design Patterns:

After programming several projects using object orientation, you may find that you are programming similar structures again and again. In the book *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, the authors describe 23 design patterns that they have found to be recurring in object orientated programming. The four authors of *Design Patterns* are nicknamed the Gang of Four (GoF) and the design patterns in the book are considered to be GoF design patterns.

Design patterns are an aid in programming in that they allow programmers to reuse a lot of code and structure instead of having to continually recode similar designs. This leads to faster development and higher-quality code.

Design Pattern Categories:

The book *Design Patterns* has three categories of design patterns. They are

- Creational Patterns
- Structural Patterns
- Behavioral Patterns

[1] This work was funded by an IBM Eclipse Innovation Grant.

[2] © Raphael Enns and David Scuse

Creational patterns involve different ways of instantiating objects. They keep the rest of the program separate from how objects are created and represented. The GoF creational patterns are Abstract Factory, Builder, Factory Method, Prototype, and Singleton.

Structural patterns describe how different classes are structured together. This involves how different classes are able to access each other and how the hierarchy of objects is set up. The GoF structural patterns are Adapter, Bridge, Composite, Decorator, Façade, Flyweight, and Proxy.

Behavioral patterns involve how objects interact with each other. This primarily involves splitting up functionality between classes and accessing this functionality without necessarily knowing how the system is structured. The GoF behavioral patterns are Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, and Visitor.

Although new categories have been formed in addition to the three mentioned above, this tutorial will only look at a selection of GoF design patterns that are in the three categories mentioned above.

Singleton:

The singleton design pattern is a creational pattern and is one of the simplest design patterns. A singleton class will only allow one instance of itself to be instantiated. It checks whether an instance has been created already and if so returns that instance. Shown below is the basic structure of a singleton class.

```
public class Singleton
{
	private static Singleton instance;

	private Singleton()
	{
	}

	public static Singleton getInstance()
	{
		if (instance == null)
			instance = new Singleton();
		return instance;
	}
}
```

In the singleton class there is a private static field of the singleton instance. The null constructor is made private so that a calling class cannot use the constructor to create an instance. The only way to get an instance of the singleton class is through the public static getInstance() method. The calling class would get the singleton instance with the following line:

```
Singleton instance = Singleton.getInstance();
```

The getInstance() method checks to see if an instance has been created already by testing if its instance field is null. If it is null, an instance has not yet been created so the instance field is then instantiated. If it is not null, an instance has already been created. The instance field is then returned. Once an instance has been created, the same instance will always be returned by the getInstance() method.

A slightly different way of implementing a singleton class is to change the field declaration to

```
private static final Singleton instance = new Singleton();
```

Using this approach, an instance of the singleton class is always created. The test to see if the instance has been created can then be removed so that the only thing the getInstance() method does is return the instance field.

Factory Method:

The factory method design pattern is a creational pattern that uses the factory principle. The factory principle involves a method which creates an instance of an object and then returns the object. Parameters may be passed to the method which can be used to select the type of object to return.

The factory method design pattern uses inheritance to determine the correct object to instantiate. A base creator class is created that may be abstract. Concrete creator classes extend the creator class. The factory method is declared in the creator class. If the factory method is abstract, all of the creator class' subclasses must also have that method. If the factory method is not abstract, the subclasses may override it if necessary.

A class may make an instance of one of the concrete creator classes. Then the class can call the concrete creator class' factory method to get the appropriate object.

For an example, suppose we have two classes we want to make instances of, Product1 and Product2. To be able to receive an instance of either of these two classes, they must both extend or implement the same class or interface. We will create an interface called Product that Product1 and Product2 will implement.

Next we will create the base creator class. We will create an abstract class called Creator, which is shown below. It will have one abstract method called getProduct().

```
public abstract class Creator
{
        public abstract Product getProduct();
}
```

We will now create two classes that extend Creator. We will call these classes Product1Creator and Product2Creator. Each one has a getProduct() method that returns the appropriate Product type. Product1Creator is shown below and Product2Creator is identical except the getProduct() method returns a Product2 object.

```
public class Product1Creator extends Creator
{
        public Product getProduct()
        {
                return new Product1();
        }
}
```

To create a new Product, a class must first have an instance of either Product1Creator or Product2Creator. Then they simply call the getProduct() method of that instance and they will receive the appropriate object.

In the example above there is a one-to-one mapping between the creator classes and the product classes. This need not be the case. Several creator classes may return the same product class or a creator class may return several types of product classes, depending on parameters passed.

The factory method design pattern is used when a class is unable to anticipate the exact object it must create.

Adapter:

The adapter design pattern is a structural pattern that allows classes to work together even though they have incompatible interfaces. The adapter pattern converts one interface into another interface that is required by the associated class.

The adapter design pattern is often used in the following scenario. You have a class already created that you wish to use. However, the application does not expect the interface of your class. It may expect an entirely different interface. To allow the application to use your class, an adapter needs to be made. The adapter receives requests from the application and then converts those requests to how your class is set up. In this way, the application thinks it is using a compatible class and you are able to use your class.

To use the adapter pattern, you need to create an interface for the expected type. This interface is called the target interface. In our target interface shown below we have two methods that need to be implemented, add(String s) and remove(String s). These are the methods that the application expects to see.

```
public interface Target
{
        public void add(String s);
        public void remove(String s);
}
```

The incompatible class that you wish to use is called the adaptee class. For the example we will say that our adaptee class has two methods, addItem(String s) and deleteItem(String s), that add and delete elements from a vector. The adaptee class is shown below.

```
import java.util.Vector;

public class Adaptee
{
        private Vector v;

        public Adaptee()
        {
                v = new Vector();
        }

        public void addItem(String s)
        {
                v.addElement(s);
        }

        public void deleteItem(String s)
        {
                v.removeElement(s);
        }
}
```

To use the adaptee class, even though the application expects the add() and remove() methods, we create an adapter class that implements the target interface. The adapter class creates an instance of the adaptee class and when the target methods are called, the appropriate adaptee methods are called by the adapter. The adapter class is shown below.

```
public class Adapter implements Target
{
        private Adaptee adaptee;

        public Adapter(Adaptee adaptee)
        {
                this.adaptee = adaptee;
        }

        public void add(String s)
        {
                adaptee.addItem(s);
        }

        public void remove(String s)
        {
                adaptee.deleteItem(s);
        }
}
```

Now if you want to use your class in your application, create an instance of the adapter and then call the methods that the application requires.

The type of adapter shown above is the object adapter. In a slightly different form of adapter, called the class adapter, the adapter class extends the adaptee class and then implements the target interface. Since the adapter is a subclass of the adaptee, it can access its methods without having to first create an instance of the adaptee.

Another common use of adapters is when you wish to implement an interface but do not need all parts of it. This often occurs while using event listeners in GUI design. For example, if you wish to implement the SelectionListener interface in SWT, you have to implement the methods widgetSelected() and widgetDefaultSelected() even if you only want to use one of them. If you use SelectionAdapter instead of SelectionListener, you only have to implement the methods that you want because the SelectionAdapter implements the SelectionListener and creates empty methods for widgetSelected() and widgetDefaultSelected().

Composite:

The composite design pattern is a structural pattern and is used when you want objects to be either components or containers that can hold other components. This is similar to a tree design where the leaf nodes are the components and the non-leaf nodes are the containers. A common place to find the use of composites is in GUI widgets. SWT has a widget called composite which is able to hold other widgets, including other composites.

Composites are able to be created in several ways. Sometimes you may allow leaf nodes to become non-leaf nodes by adding children to them, though usually leaf nodes are restricted from having children. You can also set up the structure of inheritance in different ways.

In our example we will restrict the non-leaf nodes from having children. We will have an abstract component class with a concrete component class that extends it. The composite class extends the concrete component class.

```
import java.util.NoSuchElementException;
import java.util.Vector;

public abstract class AbstractComponent
{
        protected AbstractComponent parent = null;
        protected Vector children = new Vector();

        public AbstractComponent getParent()
        {return parent;}

        public abstract AbstractComponent getChild(int index)
                throws NoSuchElementException;
        public abstract void addChild(AbstractComponent child)
                throws NoSuchElementException;
        public abstract void removeChild(AbstractComponent child)
                throws NoSuchElementException;
}
```

Shown above is the abstract component class. It has a vector to hold the children, a parent, a method to get the parent and several abstract methods that are for performing operations on an object's children. Since we want to restrict the leaf nodes from having children, we add throwable exceptions to the abstract methods.

The component class extends the abstract component class. Since components are restricted from having children, any methods that perform operations on children throw an exception. Below is shown a basic component class.

```
import java.util.NoSuchElementException;

public class Component extends AbstractComponent
{
        public void addChild(AbstractComponent child)
                throws NoSuchElementException
        {
                throw new NoSuchElementException("Cannot add child to leaf");
        }

        public void removeChild(AbstractComponent child)
                throws NoSuchElementException
        {
                throw new NoSuchElementException("Cannot remove child from leaf");
        }

        public AbstractComponent getChild(int index)
                throws NoSuchElementException
        {
                throw new NoSuchElementException("Cannot get child from leaf");
        }
}
```

The composite class extends the concrete component class. Composites are allowed to have children, so the methods performing operations on children are properly implemented. Below is shown a basic composite class.

```
import java.util.NoSuchElementException;

public class Composite extends Component
{
        public void addChild(AbstractComponent child)
                throws NoSuchElementException
        {
                children.addElement(child);
        }

        public void removeChild(AbstractComponent child)
                throws NoSuchElementException
        {
                children.removeElement(child);
        }

        public AbstractComponent getChild(int index)
                throws NoSuchElementException
        {
                return (AbstractComponent) children.elementAt(index);
        }
}
```

To add individual components and composites to the example above, simply make classes that extend the component and composite classes. In a real program, you would probably have many more fields and methods in your classes to make them functional.

Another way of structuring the composite design pattern is to have an abstract class for the components and an abstract class for the composites that extends the abstract component class. Then all components would extend the abstract component class and all composites would extend the abstract composite class.

Command:

The command design pattern is a behavioral pattern that allows you to create a command as an object and then execute it as necessary. This pattern is useful if you want to create undo / redo operations, log operations, or execute commands at a later time.

A command interface is necessary for this pattern. For an object to be a command object, it must implement a method that will execute its command. Shown below is a simple command interface with one method that needs to be implemented, execute().

```
public interface Command
{
        public void execute();
}
```

Any classes that are to be commands must now implement the command interface. Below is shown a simple command that prints "Hello World!" to the console when the command is executed.

```
public class HelloCommand implements Command
{
        public void execute()
        {
                System.out.println("Hello World!");
        }
}
```

You can execute this command by creating an instance of HelloCommand and then by calling its execute method.

To create a history of commands that would be useful for undo / redo and logging operations, you would need to have a command manager or command handler class. This class would receive all the commands, save the command in a history data structure, and then execute the command. The history could then be used to undo and redo operations or to resume operations up to the current point in case of a crash.

Iterator:

The iterator design pattern is a behavioral pattern that iterates through a data structure. An iterator traverses a data structure and gives access to the data structure's elements without revealing the data structure's internal representation. It also allows you to traverse a data structure in several ways without bloating the data structure's interface.

An iterator has a next element method and often has a method to check if the end of the traversal has been reached. All of the traversal details can be hidden from the client so that one method call is all that is needed to get the next element. If the data structure is changed when using an iterator, only the iterator will have to be changed. The calling classes will not need to be modified.

An iterator needs an iterator interface. Java already comes with two basic iterator interfaces that you can implement. They are called Iterator and Enumeration. Both are in the java.util package. Several data structures built into Java such as lists use one or both of these iterators. Listed below is the code from each of the iterators.

```
public interface Iterator
{
        public boolean hasNext();
        public Object next();
        public void remove();
}

public interface Enumeration
{
        public boolean hasMoreElements();
        public Object nextElement();
}
```

As you can see, both iterators have a method that checks if there are more elements, and both have a method that gets the next element. The Iterator interface, however, has one more method than the Enumeration interface. The remove() method removes that last received object from the data structure and can only be called once for each call to next().

If you need an iterator that provides more functionality than the basic iterators provided for you by Java, you can simply write your own. Perhaps you would like several types of traversals possible. An example is if you want to traverse a list from the bottom or from the top.

As an example, we will create a simple iterator for a Vector based on Java's Iterator interface. Shown below is the VectorIterator class.

```java
import java.util.Iterator;
import java.util.Vector;

public class VectorIterator implements Iterator
{
	private Vector v;
	private int current;
	private boolean removeEnabled;

	public VectorIterator(Vector v)
	{
		this.v = v;
		current = 0;
		removeEnabled = false;
	}

	public boolean hasNext()
	{
		if (current >= v.size())
			return false;
		else
			return true;
	}

	public Object next()
	{
		Object result = null;

		if (hasNext())
		{
			result = v.elementAt(current);
			current++;
			removeEnabled = true;
		}

		return result;
	}

	public void remove()
	{
		if (removeEnabled)
		{
			v.removeElementAt(--current);
			removeEnabled = false;
		}
	}
}
```

The VectorIterator class implements Java’s Iterator interface, therefore it must have the methods hasNext(), next(), and remove(). One of the fields is a Vector, which is a reference to the Vector to be traversed. The field named current is the current position in the traversal. The removeEnabled flag is to allow only one call to remove() for each call to next().

The hasNext() method checks the position in the traversal against the size of the Vector. If the traversal position has reached the end of the Vector, false is returned. The next() method first makes sure there are more elements left in the traversal by calling hasNext(). Then it increments the traversal position and returns the element. The remove() method decrements the traversal position and then deletes the most recently returned element from the Vector.

To use the VectorIterator in your code, first create a Vector and then pass it to the constructor when instantiating a VectorIterator, as shown below.

```
Vector v = new Vector();
//add some elements to the Vector here
VectorIterator vi = new VectorIterator(v);
```

Now you can iterate through the vector by calling the VectorIterator instance's next() method.

Template Method:

The template method design pattern is a behavioral pattern that extracts common elements of program flow and places them in a superclass. This is a simple pattern that is used all the time. The template method pattern is used when common elements are placed in a base class where subclasses are used to implement the elements that vary.

The base class in the template method pattern is often abstract. It will have one or more methods that are either abstract or have a default implementation that can be overridden by the subclasses.

For an example, we will use the template method pattern to create classes to calculate interest. Interest is calculated two ways, with simple interest or with compound interest. The formula for simple interest is $A = P(1 + ni)$ and the formula for compound interest is $A = P(1 + i/q)^{nq}$, where A is the total amount, P is the principal amount, n is the number of years, i is the interest rate, and q is the number of times interest is compounded per year.

We will create an abstract base class called Interest that has a method getAmount() that returns the total amount. The getAmount() method returns the product of the principal and the rest of the formula which is calculated by the subclasses in the abstract method calcInterest(). The Interest base class is shown below.

```
public abstract class Interest
{
	private double principal;

	public Interest(double principal)
	{
		this.principal = principal;
	}

	public double getAmount()
	{
		return principal * calcInterest();
	}

	protected abstract double calcInterest();
}
```

To calculate the simple interest, we create a class SimpleInterest that extends Interest.

```
public class SimpleInterest extends Interest
{
        private int numYears;
        private double interestRate;

        public SimpleInterest (double principal, int numYears,
                double interestRate)
        {
                super(principal);
                this.numYears = numYears;
                this.interestRate = interestRate;
        }

        protected double calcInterest()
        {
                return (1 + numYears * interestRate);
        }
}
```

To calculate the compound interest, we create a class CompoundInterest that extends Interest.

```
public class CompoundInterest extends Interest
{
        private int numYears;
        private double interestRate;
        private int compoundsPerYear;

        public CompoundInterest (double principal, int numYears,
                double interestRate, int compoundsPerYear)
        {
                super(principal);
                this.numYears = numYears;
                this.interestRate = interestRate;
                this.compoundsPerYear = compoundsPerYear;
        }

        protected double calcInterest()
        {
                return Math.pow(1 + interestRate / compoundsPerYear,
                        numYears * compoundsPerYear);
        }
}
```

Both classes above implement the calcInterest() method where they calculate the part of the formula that is unique to each kind of interest.

As you can see, the template method design pattern is a fairly simple idea and you have probably used it often whether you knew it or not.

Using CodePro to Create Design Patterns:

Although you can learn all the details of the design patterns and then implement them when you need them, it is often easier to use a program to automatically generate the basic structure of the patterns for you. A suite of Java development tools called CodePro contains wizards for design patterns as well as numerous other development tools such as metrics and code analyzing.

CodePro is a commercial plug-in for Eclipse. There are several versions of CodePro. CodePro is split up into three parts, Advisor, Agility, and Build, each of which can be purchased separately. The Studio version contains all three parts as well as some collaboration tools. CodePro Express contains limited features from all three parts of CodePro. CodePro Studio supports all of the 23 GoF design patterns as well as several other patterns. CodePro is made by Instantiations, Inc. Their website is http://www.instantiations.com.

Creating design patterns using CodePro is easy. Under the File menu, select New > Other. Select Java on the left side and Pattern on the right side. A window appears with all the design patterns that CodePro can make. Select the design pattern you wish to create and click on Next. One or more wizard pages will then appear allowing you to customize the selected pattern.

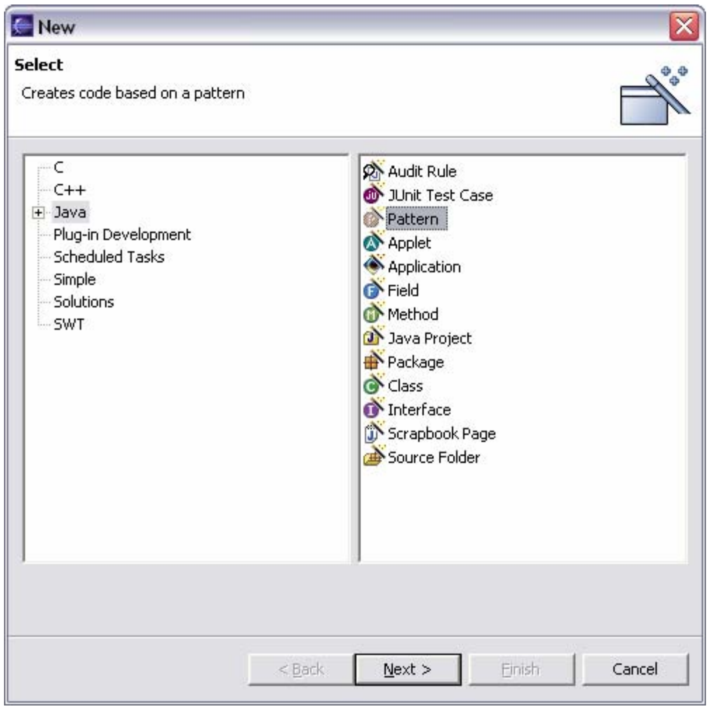
You can also easily add your own design patterns to those that CodePro already supports. For more information on adding design patterns to CodePro, see the documentation at http://www.instantiations.com/codepro/ws/docs/features/patterns/adding_new_patterns.html.

The documentation for CodePro at Instantiations' website can be found at http://www.instantiations.com/codepro/ws/docs/default.htm. CodePro is well documented, so we will not look at it in depth in this tutorial. Each design pattern has its own page in the documentation describing the pattern as well as any wizard pages for the design pattern. The documentation pages for each pattern also have links to several websites that have information on the design pattern.

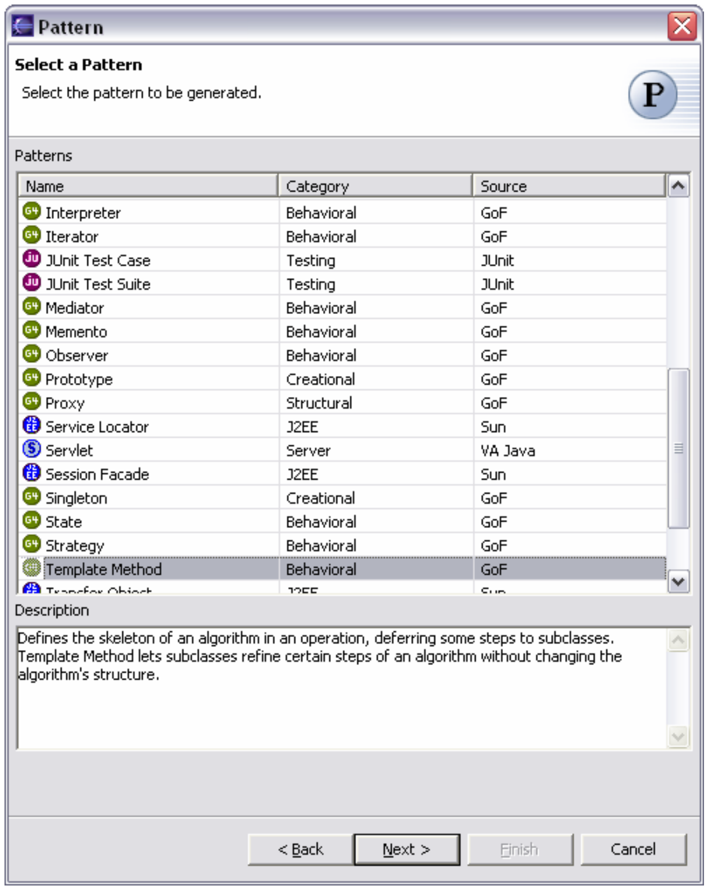
CodePro Example:

To show how easy it is to create the structure for a design pattern using CodePro, we will use CodePro to create the example we used in the template method section of this tutorial.

First, select New > Other from the File menu. Select Java on the left and Pattern on the right. Below is shown the selection window.



Clicking on Next brings you to the following window.

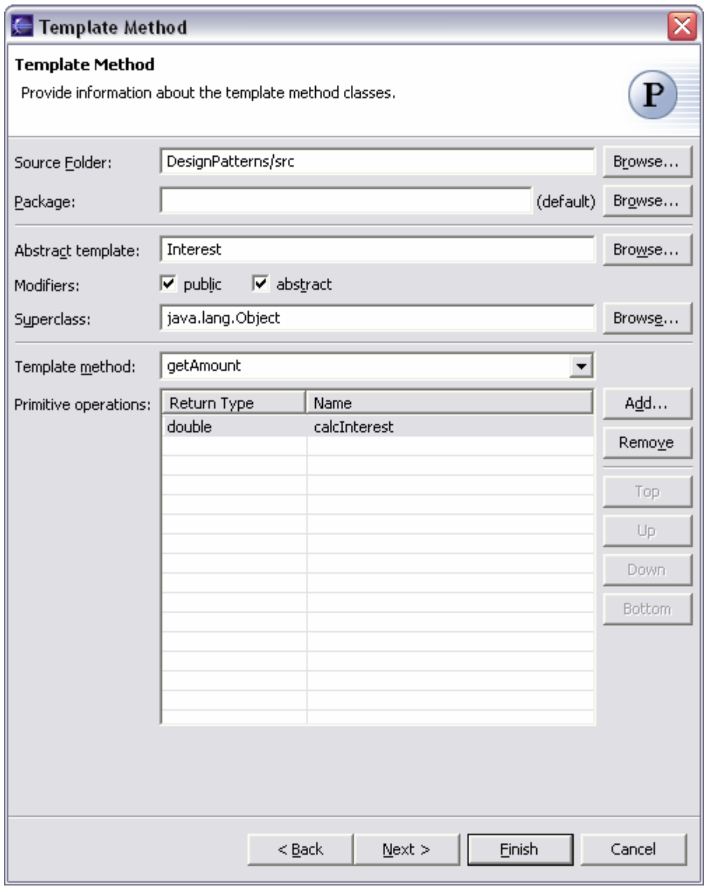


Name	Category	Source
Interpreter	Behavioral	GoF
Iterator	Behavioral	GoF
JUnit Test Case	Testing	JUnit
JUnit Test Suite	Testing	JUnit
Mediator	Behavioral	GoF
Memento	Behavioral	GoF
Observer	Behavioral	GoF
Prototype	Creational	GoF
Proxy	Structural	GoF
Service Locator	J2EE	Sun
Servlet	Server	VA Java
Session Facade	J2EE	Sun
Singleton	Creational	GoF
State	Behavioral	GoF
Strategy	Behavioral	GoF
Template Method	Behavioral	GoF
Transfer Object	J2EE	Sun

Description

Defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses refine certain steps of an algorithm without changing the algorithm's structure.

Select Template Method from the list and click Next.


Template Method

Template Method

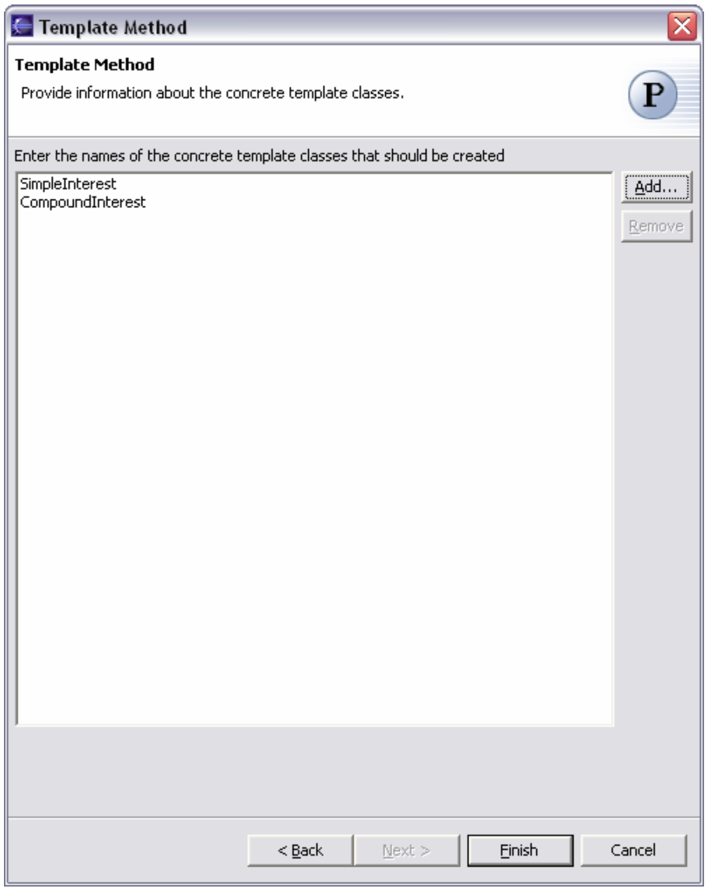
Provide information about the template method classes.

Source Folder:	DesignPatterns/src	Browse...
Package:	(default)	Browse...
Abstract template:	Interest	Browse...
Modifiers:	☑ public ☑ abstract	
Superclass:	java.lang.Object	Browse...
Template method:	getAmount	

Primitive operations:

Return Type	Name
double	calcInterest

In the first page of the Template Method wizard, set the Abstract template to "Interest", set Template method to "getAmount", and then add a Primitive operation by clicking on the Add button. To change the Return Type and Name columns in the Primitive operations table, simply click on the cell you wish to modify and then modify it. Change the name of the primitive operation you added to "calcInterest" and change its return type to a double. Click Next to get the next Template Method wizard page which is shown below.


Template Method

Template Method

Provide information about the concrete template classes.

Enter the names of the concrete template classes that should be created

SimpleInterest
CompoundInterest

To create the subclasses that extend the abstract base class, click on the Add button. In the input dialog that appears, enter the name of the two classes, SimpleInterest and CompoundInterest, separated by a comma. Click Finish to generate your new classes.

If you look at the classes that were generated, you will see that the generated code is very similar to the skeleton of the classes we made before, along with comments that CodePro adds. To make the generated classes function like the classes we made, simply create the same constructors and fields from the previous example. In the Interest class, change the return value of the getAmount() method to a double and copy the method's code from our previous example. In the SimpleInterest and CompoundInterest classes, simply copy the code from the calcInterest() methods from our previous example to the CodePro generated classes.

Now the CodePro generated classes are identical to the classes that previously we had to create all by ourselves. We were able to save ourselves time by letting CodePro generate the structure for the design pattern. All we have to do afterwards is put in the details that are specific to our implementation of the pattern. As you can see, using a design pattern code generator tool such as CodePro can save a significant amount of time.

Summary:

You have seen several design patterns in this tutorial, as well as some simple examples that show how they are structured. By applying these design patterns in your system design, you will have code that is more polymorphic in nature, easier to add on to, and easier to reuse.

This tutorial has given a brief description of only a few of the many design patterns. There have been several books written on design patterns and there are numerous websites that describe the different design patterns. Look at these resources for more information on design patterns. A website that describes a number of design patterns can be found at http://www.patterndigest.com.

We looked at a program called CodePro that generates code for design patterns. CodePro is a suite of Java development tools made by Instantiations, Inc. For more information on what CodePro is, what it can do, and where to purchase it, go to Instantiations' website at http://www.instantiations.com.