Patterns and Java

In general the original GoF patterns and other OO patterns can be translated into Java without much effort.

However, Java offers some particular features, such as

- interfaces,
- package access and
- very good threading support,

as well as some specific constraints, such as lack of

- multiple inheritance and
- generics (well until Java generics are added in Java 1.5?)

that require care when implementing a pattern.

As with all usage of patterns it is important to determine the applicability of the pattern to the design and application domain independent of the language used for the implementation.

When the pattern is deemed applicable, you then need to implement it in the selected language – Java is the case of concern here.

It is also useful to identify particular idioms of the language in your implementation. These can constitute patterns themselves but generally cuts across patterns as general implementation tactics for the language (e.g. using Java interfaces to define roles).

Singleton [GoF, 127]

Intent and Motivation

Guarantees that only one instance of a particular class can exist and gives a means to access that instance.

Singletons are used in OO programs somewhat similarly with global data in functional programming (which is generally considered bad practice). A Singleton should be a class for which logically only one instance should exist.
A good rule of thumb is to ask yourself if it makes sense in the domain of your applications that many instances of the Singleton class should or could exist.

**Abstract UML Structure**

```
getInstance():SingletonClass is a class method. It constitutes the global access point to the only instance of the class.
```

Typical client interaction is as in the following sequence diagram.

```
In general, the unique instance of a Singleton is created the first time it is accessed; but that need not always be the case.

**Java Implementation**

There are two common direct Java implementation of the Singleton pattern that differs in the time of creation of the Singleton instance:

- in the first case, the `getInstance()` lazily creates the unique instance
- in the second case, the unique instance is created directly as a class instance or created using the static initializer of the class (e.g., `static { ... }`). In both cases, in Java, the instance is created when the class is loaded
Note: For either implementation, the class constructor is marked as private or protected access to prevent clients from directly creating instances.

```java
public class Singleton {
    protected Singleton() { name = "Singleton"; }

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

        return instance;
    }

    public String toString() { return name; }

    public void operation1() {
        //<temp>
        System.out.println("<operation1Code/>");
        //</temp>
    }
}
```

Java Considerations

In the lazy creation implementation, the instance is only created when it is first accessed and this means:

1. The first client thread accessing the Singleton incurs all creating costs.
2. The instance is created only when it is needed (lazily). If it is never accessed (which probably is rare) it is never created.

In the other case, the instance is created at class loading time, which occurs in Java if any instance of the class is created, passed as an argument or a class constant or class method (e.g., `getInstance()` ) is accessed.
A third alternative (more complicated) implementation in Java is to allow the `SingletonClass` instance to be specialized (that is subclassed) and use a combination of Java reflection and system properties to configure and instantiate the correct instance.

In this case the `SingletonClass` is the default implementation but different implementation subclasses can be provided and the client can decide at run time which one should be loaded.

The `SingletonClass`'s constructor can no longer be marked as private since the subclass needs to call `super()`, so it should be marked protected.

Here is a sketch of this implementation:

```java
public static DynamicSingleton getInstance( Class singletonClass )
        throws RuntimeException
{
    if( instance != null ) return instance;

    try
    {
        Constructor ctor = singletonClass.
                getConstructor( new Class[ 0 ] );
        instance = (DynamicSingleton)ctor.
                newInstance( new Object[ 0 ] );

        return instance;
    }
    catch( Exception e )
    {
        throw new RuntimeException( "The class : "
        + singletonClass.getName()
        + " does not have correct ctor" );
    }
}

public static DynamicSingleton getInstance()
{
    if( instance != null ) return instance;

    try
    {
        Properties props = new Properties();

```
```java
props.load( new FileInputStream(
    "singleton.properties" ) );

if( props.containsKey("singletonClass") )
    return getInstance( Class.forName(
        props.getProperty(
            "singletonClass" ) ) );
else
    return getDefaultInstance();
} catch( Exception e ) { return getDefaultInstance(); }
```

*Note:* Yet another approach is the have a `getInstance(Class : aClass)` where the client can decide at runtime vs. configuration time the Singleton instance subclass that it wants. Also, the `getDefaultInstance()` is implemented just like the `getInstance()` of the regular Singleton.

However, you might need to make the `getInstance(...)` method throw some exception if the class passed is not a valid concrete subclass of the Singleton class.

*Consequences*

The GoF book mentions that the Singleton not only controls access to the sole instance and restricts creation but also, unlike global variables, “avoids polluting the name space” [GoF, 129] since the instance is associated with the class name.

Since it is simple to implement, the Singleton pattern, is sometimes abused. The Singleton should not be used as a means to allow global variables in Java.

Remember to make sure that the Singleton design invariant – that is that the Singleton class logically should only have one instance – applies; this is typically an application domain restriction (e.g. a single Manager, Registry or DataStore objects).
Command [GoF, 233] and TaskScheduler

Intent and Motivation

The Command pattern “encapsulates request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and supports undoable operations.” [GoF, 233]

The Command pattern can be thought of as a functor, or an object that represents a function.

It is especially useful in languages that do not support function objects directly like Java. This pattern is especially useful for the creation of asynchronous operation as illustrated by the TaskScheduler example.

Abstract UML Structure

The abstract structure for the Command pattern quite simply comprises the Command interface and concrete implementation of that interface.

The GoF book (page 235) mentions that the Command pattern can be expanded to support macro-commands that is essentially a Composite [GoF, 163] Command, which is a Command that includes many other Command, whose execution results in the execution of all the other Command that it contains in sequence.

The diagram above illustrates the MacroCommand. The MacroCommand.execute() method is usually implemented as:
public void execute()
{
    for each command in list
        command.execute();
}

Clients usually create concrete Commands and pass them to receivers who keep them and call back on them to execute without knowledge of the details of the execution.

**Java Implementation**

The implementation of the Command pattern in Java is trivial. The Command should be implemented as a Java interface and concrete Command as classes that implement the interface.

Usually an Abstract Factory [GoF, 87] is provided to allow clients to create the concrete Command objects.

To illustrate the Command pattern, we will describe another pattern that makes use of the Command pattern to allow the creation and execution of asynchronous calls and schedule them in a scheduler.

This is a **TaskScheduler** and the **Task** is the Command pattern. The implementation also demonstrates a simple but common application of concurrent programming (threading) in Java [Lea00].

The UML structure for the **TaskScheduler** is as follows:
For simplicity the Queue interface implementation is not shown – left as an exercise for the reader.

The FifoTaskScheduler uses the queue to keep a list of posted Task to execute. It has an internal Java Thread object that waits on a lock until a new Task is posted at which time the Thread execute each Task in the Queue until it is empty.

The run() method follows Lea’s Event Queue (page 295) example and just uses a simple Object lock to synchronize access to the scheduler and notify the worker thread that some new Tasks are posted. Sketches of the different classes are shown below:

```java
public interface Task
{
    public void execute();
    public boolean isInException();
    public Exception getException();
}

public interface Queue
{
    public boolean isEmpty();
    public void enqueue( Object element );
    public Object dequeue() throws NoSuchElementException;
}
```
public interface TaskScheduler
{
    public void post( Task task );
    public boolean isStopped();
    public void start();
    public void stop();
}

public abstract class AbstractTaskScheduler extends Object implements TaskScheduler
{
    public AbstractTaskScheduler() {}  
    public abstract void post( Task task );

    public boolean isStopped()
    { return ( started == false ); }  

    public void start()
    {
        if( started ) return;
        started = true;
        stopped = false;
        synchronized( startLock ) { startLock.notifyAll(); }
    }

    public void stop()
    {
        if( stopped ) return;
        stopped = true;
        started = false;
        synchronized( stopLock ) { stopLock.notifyAll(); }
    }

    // Instance variable

    protected boolean started = false;
    protected boolean stopped = true;
    protected Object startLock = new Object();
    protected Object stopLock = new Object();
}
public class FifoScheduler extends AbstractTaskScheduler {
    public FifoScheduler() { start(); }

    public void post( Task task )
    {
        if( isStopped() )
            throw new RuntimeException( "Cannot post( Task )" +
                + "to a stopped TaskScheduler" );
        queue.enqueue(task);
        synchronized( postLock ) { postLock.notifyAll(); }
    }

    public void start()
    {
        if( started ) return;
        started = true;
        stopped = false;
        queue = new DefaultQueue();
        activeObject = this.new ActiveObject();
        synchronized( startLock ) { startLock.notifyAll(); }
    }

    private void runActiveObject()
    {
        while( true )
        {
            //If we are not yet stated then wait
            while( started == false )
                synchronized( startLock )
                {
                    try{ startLock.wait(); } catch( InterruptedException ie ) {}}

            //Check if queue is empty and wait
            //otherwise execute task
            while( queue.isEmpty() )
                synchronized( postLock )
                {
                    try{ postLock.wait(); } catch( InterruptedException ie ) {}}
                }
            executeTask( (Task)queue.dequeue() );
        }
}
// Check to see if we are stopped
if( stopped == true )
{
    started = false;
    return;
}
}

private void executeTask( Task task )
{
    try{ task.execute(); }
    catch( Exception e ) {}}

class ActiveObject extends Object implements Runnable
{
    public ActiveObject()
    {
        thread = new Thread( this );
        thread.setName( "FifoScheduler.ActiveObject" );
        thread.start();
    }

    public void run()
    { FifoScheduler.this.runActiveObject(); } }

    private Thread thread = null;
}

private Object postLock = new Object();
private Queue queue = null;
private ActiveObject activeObject = null;

Java Considerations
The Command pattern with a simple execute() method lacks means for error handling, that is, if an exception occurs during the execute method, the client needs to have a way to know that this occurred and also a way to retrieve the Exception. The Task interface solves this problem by adding an isInException():boolean method as well as a getException():Exception method.
**Consequences**

The Command objects are a nice way to decouple the invoker of an operation from the object that actually performs it. The GoF book also mentions that Command objects are “first-class objects” and can be extended and manipulated like other objects in the application. Command objects can also be aggregated into MacroCommand – essentially Composite [GoF, 163] commands.
Bridge [GoF, 151]

Intent and Motivation

“Decouples an abstraction from its implementation so that the two can vary independently.” [GoF, 151]

The bridge pattern allows the implementation of some abstraction to vary or be implemented in various ways.

This is useful whenever you want to create portable abstractions and can foresee various possible implementations of these abstractions using different libraries for example.

Abstract UML Structure

The abstraction can be one class or a hierarchy of classes. For each abstraction class in the hierarchy there is a correspondent implementation class.

The implementation classes can be also abstracted as a hierarchy of abstract classes that parallels the abstraction hierarchy. Concrete implementation classes mirrors the abstraction classes and implement the correct abstract implementation class.
The bridge pattern works simply by having each abstraction class forward any client call to its implementation. The abstraction class can also do some pre and post processing for each method call\(^1\).

**Java Implementation**

To illustrate the Java implementation of the bridge pattern we will show a simple logging abstraction that can have various implementations.

The motivation for the example is that there are a few logging facilities for Java (Log4J from Apache, jLog and the JCP JSR47 logging API that will be included in Java 1.4) to avoid coupling an application to any particular implementation, we create a simple abstract `Logger` and use the Bridge pattern to allow its implementation to vary depending on which logging API is used on the system.

In the code below we only show a concrete file logging implementation but the other implementations should similarly follow.

Note also that the logging abstraction is simplified to make the example short and to concentrate on illustrating the Bridge pattern rather than how to build a logging facility.

\(^1\) This is nice way of doing programming by contract – as found in languages like Eiffel – in Java. Note however, that there exist pre-processing tools that help to add contract to Java code, such as the iContract tool by Reto Kramer (http://www.reliable-systems.com).
public class LogEntry extends Object
{
    public LogEntry(String message, int priority)
    { this.message = message; this.priority = priority; }

    public String getMessage() { return message; }
    public int getPriority() { return priority; }

    public String toString()
    {
        StringBuffer sb = new StringBuffer();
        sb.append( "<LogEntry>\n" );
        sb.append( "<\tmessage=" + getMessage() + "\t>\n" );
        sb.append( "<\tpriority=" + getPriority() + "\t>\n" );
        sb.append( "</LogEntry>\n" );
        return sb.toString();
    }

    private String message = "";
    private int priority = 0;

    public static final int PRIORITY_INFO = 0;
    public static final int PRIORITY_NORMAL = 1;
    public static final int PRIORITY_CRITICAL = 2;
}

public class Logger extends Object
{
    public Logger(LoggerImp loggerImp)
    { this.loggerImp = loggerImp; }

    public void add(LogEntry entry)
    { loggerImp.add(entry); }

    public void flush() throws Exception
    { loggerImp.flush(); }

    private LoggerImp loggerImp = null;
}
import java.util.*;
import java.io.*;

public class FileLoggerImp extends Object implements LoggerImp
{
    private String logFileName = "";
    private List entryList = new ArrayList();

    public FileLoggerImp(String fileName)
    {
        this.logFileName = fileName;
    }

    public void add(LogEntry entry)
    {
        entryList.add(entry);
    }

    public void flush() throws Exception
    {
        FileOutputStream fos =
            new FileOutputStream(logFileName, true);
        Iterator iterator = entryList.iterator();
        while (iterator.hasNext())
        {
            String entryString = iterator.next().toString() + "\n";
            fos.write(entryString.getBytes());
        }
        fos.close();
        entryList.clear();
    }

    public class FileLoggerImp
    {
        public FileLoggerImp(String fileName)
        {
            this.logFileName = fileName;
        }

        public void add(LogEntry entry)
        {
            entryList.add(entry);
        }

        public void flush() throws Exception
        {
            FileOutputStream fos =
                new FileOutputStream(logFileName, true);
            Iterator iterator = entryList.iterator();
            while (iterator.hasNext())
            {
                String entryString = iterator.next().toString() + "\n";
                fos.write(entryString.getBytes());
            }
            fos.close();
            entryList.clear();
        }
    }

Java Considerations

The key consideration for implementing the Bridge pattern in Java is whether the abstract implementation hierarchy should be abstract classes or interfaces.

If they are abstract classes then common implementation code can be provided – note however that common implementation code can also be provided in the abstraction side – and on the other hand, if they are a hierarchy of interfaces then the concrete implementation can inherit from other classes – that is they are not forced to implement the abstract implementation classes.

Another consideration is that the abstraction classes’ method should usually have the same signature – including exception clause – as the implementation methods. This makes the method implementation as simple as:
public void operation( <Arguments> ) throws Exception
{
    //Do pre-processing
    getImp().operation( <Arguments> );
    //Do post-processing if no exception
}

Consequences

Some consequences of the bridge pattern are as follows:

1. The abstraction and implementation classes need to be wired together. This is best done via an Abstract Factory [GoF, 87] that can hide the details of creating the concrete implementation classes and then the abstraction classes and wiring them together. Both the abstraction and implementation constructor cannot expect the other object as argument since this would result into a circular dependency.

2. The abstract implementation hierarchy classes can have other methods in them especially if some abstraction method can be partially implemented with some algorithm and defer some steps to the implementation – this is similar to the application of the Template Method [GoF, 325] pattern.

3. The GoF book mentions that besides decoupling the abstraction from the implementation and allowing both to vary independently, the Bridge pattern also hides the details of the implementation from the clients that only deal with the abstraction classes.
Visitor [GoF, 331]

Intent and Motivation

The Visitor pattern allows the addition and execution of methods to a hierarchy of classes without modification to any class of the hierarchy.

The Visitor pattern “abstracts class behavior beyond class boundaries” [Jézéquel00, 228] by allowing clients of a class to define methods on the class without directly modifying the class source code or subclassing it.

Often in OO design various program abstractions are modeled into hierarchies that are fairly stable. However, it might be hard to come up a priori with all the possible operations that all clients will be applying to this hierarchy.

If the client wants to apply a method to the hierarchy then the client code typically end up with case statements or if-else statements on the object types to define the operations, as for example (in Java):

```java
public void method( SomeType object )
{
    if( object instanceof SomeTypeA )
      someType1Operation();
    else
      if( object instanceof SomeTypeB )
        someType2Operation();
      else
        if( object instanceof SomeTypeC )
          someType3Operation();
        else
          someTypeOperation();  //default
}
```

This kind of conditional structure is bad because:

1. It is hard to maintain and prone to errors – especially when new types are added.
2. More importantly should be replaced by a polymorphic `operation()` method on the class hierarchy (`operation()` can be overridden at each level of hierarchy to implement the correct behavior).
Of course this is only possible if the client has access to the source code for the hierarchy. Another problem is that it might be hard to anticipate what operations will be needed and there could be many such operations – e.g. for class libraries.

The Visitor pattern solves this problem elegantly with minor overhead. The general UML structure for the Visitor pattern is as follows.

*Abstract structure*

The diagram below shows the general class structure for the Visitor pattern. As can be seen the Visitor pattern has some overhead that can be summarized in 3 steps:

1. Create an abstract type whose name is, by convention, prepended with Visitor – `SomeTypeVisitor` in the diagram
2. Add a `visit<Xyz>(xyz: Xyz):void` method for each Xyz type in the hierarchy – these are the `visitSomeType<A/B/C>(someType: SomeType<A/B/C>):void` in the diagram (parameter not shown)
3. Add an `accept(visitor: Visitor):void` in the root of the hierarchy and implement in each subclass by calling the correct `visit<Xyz>(...)` method of the visitor parameter – shown in the note below the diagram.

![Diagram](image)

Each `visitSomeType<A/B/C>()` method take as parameter an instance of `SomeType<A/B/C>` which is not shown on diagram. And also, each `SomeType<A/B/C>` implement the `accept(...)` method calling the correct `visitSomeType<A/B/C>(...)` method.
To use the Visitor pattern, the client will create a concrete subclass of the Visitor abstract type. Implement each visit method and follow the following sequence.

If the operation that the `ConcreteSomeTypeV` Visitor class represents need to return a result then you can add a method like `get<Result>()` that can be called by the client to retrieve the result once the Visitor instance has been accepted.

**Java Implementation**

The Java implementation is fairly straightforward:

1. The Visitor abstract class is best implemented as a Java interface
2. The superclass of the hierarchy usually either makes the `accept(...)` method abstract or default the implementation by calling the `visitsomeType(...)` method

The following code snippet is a contrive example showing the Visitor pattern applied to a simple `Employee, Manager, Engineer, SoftwareEng` class hierarchy for a simple company application.

`Engineer` and `Manager` extend `Employee` and `SoftwareEng` extends `Engineer`. In the end we show a concrete Visitor class to retrieve the average overtime for any instance of the hierarchy and a simple driver client class to exercise the Visitor.
public abstract class Employee
{
    public Employee( String firstName,
                     String lastName,
                     long employeeNumber )
    {
        this.firstName = firstName;
        this.lastName = lastName;
        this.employeeNumber = employeeNumber;
    }

    public abstract String getJobName();

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public long getEmployeeNumber() { return employeeNumber; }

    public void accept( EmployeeVisitor visitor )
    { visitor.visitEmployee( this ); }

    //-------------------------------
    // Instance variables
    private String firstName = "";
    private String lastName = "";
    private long employeeNumber = 0;
}

public class Manager extends Employee
{
    public Manager( String firstName,
                     String lastName,
                     long employeeNumber )
    { super( firstName, lastName, employeeNumber ); }

    public String getJobName() { return "Manager"; }

    public void accept( EmployeeVisitor visitor )
    { visitor.visitManager( this ); }
}

Similarly to the Manager class the Engineer and SoftwareEng classes are implemented with the accept method calling the correct visit<Xyz>(...) method and returning the correct JobName String.
```java
public class AverageOvertimeV extends Object implements EmployeeVisitor {

    public void visitEmployee(Employee employee) {
        averageOvertime = 0;
    }

    public void visitManager(Manager manager) {
        visitEmployee(manager);
    }

    public void visitEngineer(Engineer engineer) {
        averageOvertime = 5;
    }

    public void visitSoftwareEng(SoftwareEng softwareEng) {
        averageOvertime = 10;
    }

    public int getResult() {
        return averageOvertime;
    }

    private int averageOvertime = 0;
}

public class VisitorTest {

    public static void main(String[] args) {

        Employee[] employees = new Employee[3];
        employees[0] = new Manager("John", "Doe", 456);
        employees[1] = new Engineer("Joe", "Williams", 789);
        employees[2] = new SoftwareEng("Mike", "Max", 345);

        AverageOvertimeV visitor = new AverageOvertimeV();

        for (int i = 0; i < employees.length; ++i) {
            employees[i].accept(visitor);

            System.out.println("Average OT for "+
                        employees[i].getFirstName() +
                        " who is a " +
                        employees[i].getJobName() +
                        " is " +
                        visitor.getResult() );
        }
    }
}

The output of running the above visitorTest driver class is:
```
Average OT for John who is a Manager is 0
Average OT for Joe who is a Engineer is 5
Average OT for Mike who is a SoftwareEng is 10

Java Considerations

When building the visitor pattern using Java one can make the following observations:

1. Use an interface for the root Visitor and provide a `Default<Xyz>V` Visitor class extending the Visitor interface with empty implementations. Concrete Visitor classes can then extend the `Default<Xyz>V` and override only the necessary methods.

2. The hierarchy elements should provide public getters to the class properties in order to make them accessible to the concrete Visitor classes. Package access is insufficient if the concrete Visitor classes can reside outside of the package of the hierarchy.

Consequences

The Visitor pattern is very powerful but also somewhat difficult pattern to grasp at first.

The way to think about the Visitor is that it allows you to add methods to a hierarchy without modifying it. This is very nice but also has some important consequences:

1. The Visitor pattern necessitates some a priori setup to make it work on a hierarchy. Not only does a Visitor interface need to be provided, but each element of the hierarchy also needs to implement the `accept(...)` method to call the correct `visitXyz(...)` method.

2. Moreover, if the hierarchy is not stable, that is other leaf classes are added in future releases then a new `visitXyz(...)` method should be added to the Visitor and this implies that each concrete Visitor class need to also implement the new `visitXyz(...)` methods. This is a major drawback of the Visitor pattern.

3. Since the concrete Visitor classes are methods to the hierarchy, their implementation will usually necessitate access
to the hierarchy classes attributes. Getter and possibly setter public methods need to be provided and this can be seen as a violation of encapsulation.
References


