# Java + Patterns = Object Engineering?

**3rd Java in the Computing Curriculum Conference,
South Bank University, 25th January 1999**

**David Parsons, Systems Engineering Faculty, Southampton Institute, UK.**

## Introduction

The mid 1990s saw an explosion of interest in two separate aspects of object technology, Java and design patterns. Both 'arrived' in 1995 with the release of Java 1.0 and the book 'Design Patterns - elements of reusable object-oriented software' by Gamma, Helm, Johnson and Vlissides. The popularity of Java rested primarily on its platform independence, as well as its simplification of some of the more complex aspects of C++. The popularity of design patterns was based on the promise of design reuse at a level of granularity applicable across a wide range of application domains. The two have proved complementary tools in the ongoing search for more robust, flexible and maintainable software.

## Java and patterns in the computing curriculum

Design patterns provide a way of capturing expert practice in object-oriented design in a form that can be usefully applied to the context of teaching. They provide us with a tool for teaching quality in design at a level of granularity that students can apply in their own small-scale programs. Because they interrelate, when used together they can assist in building well-designed and coherent systems. In this respect they are a valuable contribution to the teaching of object-oriented software engineering, providing a concrete link between the activities of design and programming. Experience also suggests that such an approach invokes a positive response in students.

The language in which design patterns can be most easily recognised and applied is Java, for three reasons:

1. Java is object-oriented, and most design patterns at the coding level are object-oriented. In particular, the most influential text ("Design Patterns" by Gamma et al, 1995) is specifically a catalogue of object-oriented patterns. A number of texts have also been published that use Java as the implementation language for patterns (e.g. Cooper, 1997 and Grand, 1998).

2. An informative exercise is to identify patterns that can be seen at work in existing class libraries, and Java's libraries have the advantage of being integral and standardised. Therefore although we can perform a similar exercise in, say, C++, a number of the pre-existing patterns tend to appear in proprietary class libraries with their own non-standard characteristics.

3. Java has been developed with an awareness of patterns (particularly evident in the Abstract Windowing Toolkit), so some of the exemplars can be easily recognised, demonstrated and applied, providing a good starting point for the study of patterns and how to use them in programming.

This tutorial aims to:

- introduce design patterns
- suggest how they might be applied to the teaching of object-oriented design and programming
- identify some key patterns in Java classes
- apply patterns to the development of student programs

The objectives of the tutorial are:

- to identify patterns that appear both in Java and in the patterns literature
- to provide some Java code that instantiates a given pattern
- to identify how some simple patterns might be applied to a student-scale piece of software

Most of the patterns described here come from 'Design Patterns' by Gamma et al, and the majority of the figures are taken from the book, albeit with some modifications to make them more compatible with Java syntax and Unified Modeling Language (UML) notation. Descriptions of Java class interfaces are given in a style similar to that used by Flanagan (1998), where the method declarators are listed without their implementations, similar to a C++ class declaration or Javadoc listing.

Design Patterns and learning about design

Patterns as described by Gamma et al (and many other authors) are 'micro-architectures', small elements of software design that can be reused and combined together to build complete systems. Combinations of patterns interacting together to provide larger architectures are sometimes called 'composite patterns' (Riehle 1997), (not to be confused with the 'Composite' pattern!) The role of design patterns in the computing curriculum is twofold. First, they convey blueprints for good practice that can be directly reused, providing a straightforward way of communicating generic elements of object-oriented design. Second, they encourage a more circumspect view of how systems should be engineered, because the design patterns solution to a problem is unlikely to be the one that is immediately obvious. There is often an element of lateral thinking in the way that the patterns address particular problems, and they often provide alternatives to the obvious (but flawed) solution. Indeed a key aspect of learning and using patterns is to understand the 'anti patterns' (or 'pattern anti-examples') that provide the motivation for a particular solution (Brown et al, 1998). An anti pattern is a design that may work but is flawed in some way, typically because it is hard to maintain and extend.

The potential scope for applying patterns in the computing curriculum is very wide, since there are patterns documented for all kinds of activity, not just the small scale component levels of object-oriented programming defined here. For example Martin Fowler has documented analysis patterns (Fowler, 1997), and James Coplien has focussed on organisational patterns (Coplien, 1995). There are many pattern resources for students to exploit, including a series of PloP (Pattern Languages of Programming) conference proceedings and a number of web sites. Patterns are particularly relevant to Java programming in that it is the first language to have been developed and become popular in the patterns era.

## Section 1: Recognising patterns in Java

One good way to begin with patterns to identify where they have been used in existing software, Indeed that is exactly what patterns are - pre-existing design elements that have been used in a number of different contexts. The pattern is merely a formal documentation of the use of these design elements. This section looks at a number of patterns that can be identified in Java. Armed with the 'Design Patterns' catalogue we can see many different examples of patterns being used in the Java APIs such as Observer, Factory, and Iterator as well as various patterns in the Abstract Windowing Toolkit (AWT). Although the Gamma text covers the most commonly recurring patterns, some of those documented elsewhere are also evident in Java, such as the Reflection API.

### The Observer pattern

The Observer pattern is one of the key elements of the well-known Model View Controller (MVC) architecture that was introduced with early versions of Smalltalk. This pattern appears in figure 1, and consists of the 'Observer' and 'Subject' classes.

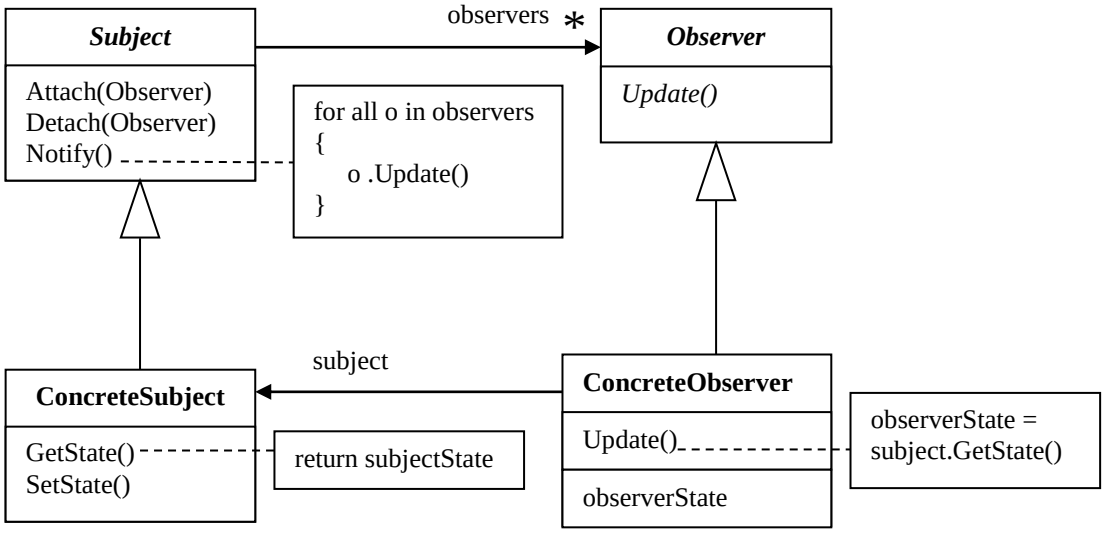


*Figure 1: The 'Observer' pattern from Gamma et al (1995)*

Java has an Observer class with an interface similar to the one in the pattern, and an 'Observable' class similar to 'Subject', though it has one or two extra methods for managing the collection of observers. The essence of the pattern is that a single subject can have multiple observers, and that changes effected to the subject via one of the observers must be reflected by all other observers updating themselves. The abstraction of this informing and updating process helps the semantic separation of an underlying model from its external views. Although this kind of pattern is most often seen in GUIs, it can be applied anywhere that the state of a number of objects is dependent on some commonly referenced object.

### The Factory Method pattern

A Factory is a method that allows an object to be created without knowing its specific class. It is also known as a 'virtual constructor', a code idiom that comes in many forms (e.g., those described in Coplien 1992). Figure 2 shows the Factory Method pattern.

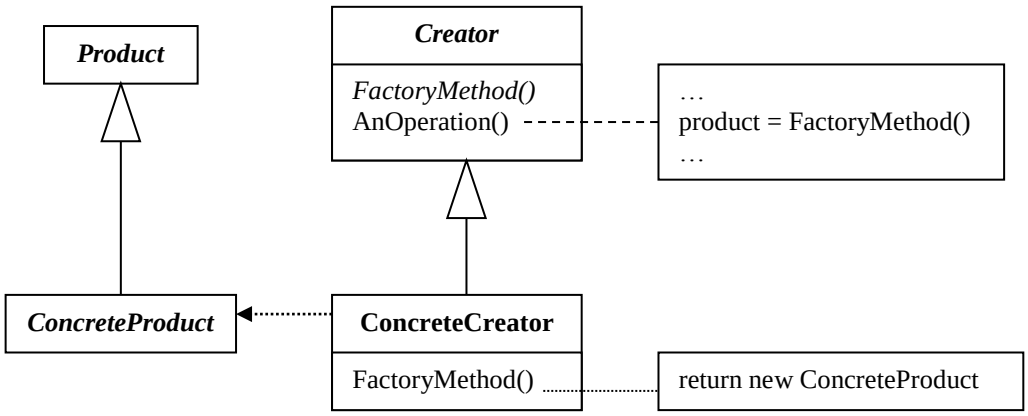


*Figure 2: The 'Factory Method' pattern from Gamma et al (1995)*

Methods that act like factories appear in a number of Java class interfaces, including Calendar and DateFormat. A factory is appropriate where the knowledge about what kind of class to be created resides with the collaborating class rather than the client.

This is the public interface of the Calendar class in Java 1.1 (The Java 2.0 class has a larger interface but makes no difference in terms of the applicability of the Factory Method pattern):

```
public abstract class Calendar
{
      void add(int, int);
      boolean after(Object);
      boolean before(Object);
      void clear();
      void clear(int);
      Object clone();
      boolean equals(Object);
      int get(int);
      int getFirstDayOfWeek();
      int getGreatestMinimum(int);
      Calendar getInstance();
      Calendar getInstance(Locale);
      Calendar getInstance(TimeZone);
      Calendar getInstance(TimeZone, Locale);
      int getLeastMaximum(int);
      int getMaximum(int);
      int  getMinimalDaysInFirstWeek();
      int getMinimum(int);
      Date getTime();
      TimeZone getTimeZone();
      boolean isLenient();
      boolean isSet(int);
      void roll(int, boolean);
      void set(int, int);
      void set(int, int, int);
      void set(int, int, int, int, int);
      void set(int, int, int, int, int, int);
```

```
        void setFirstDayOfWeek(int);
        void setLenient(boolean);
        void setMinimalDaysInFirstWeek(int);
        void setTime(Date);
        void setTimeZone(TimeZone);
}
```

This set of public methods has a number of interesting aspects. Perhaps the most significant is that there is no public constructor, so we cannot directly instantiate an object of the Calendar class. The only way we can create a Calendar object is indirectly via one of the 'getInstance' factory methods that return an object. This ensures that the appropriate type of Calendar object is created, without putting the responsibility for specifying its concrete class on the client. The actual class of the Calendar is unknown to us, since we reference it using the superclass (though in all probability it is actually an object of the GregorianCalendar class). Factory methods are useful wherever the information required to select the concrete class of an object is not readily available to the client. In some cases the selection can be made on the basis of information acquired from some other aspect of the system, such as the locale that defines the appropriate type of Calendar. In other contexts some parameter information may be supplied by the client that the factory method is able to analyse in order to return the correct class of object. One example might be an image conversion system that provides images in file formats appropriate to the current target. Since different types of software frequently have different sets of image file that they recognise, a factory method could return a concrete image format object appropriate to the target application without the client needing to specify the exact image type. The client would simply pass in the parameter information necessary for the factory method to identify the nature of the target.

The Iterator pattern

The Iterator pattern provides a standard interface for iterating through any collection of objects regardless of its own interface and implementation. It consists of two main aspects, an 'Iterator' interface with four methods and a method on an 'Aggregate' (container object) to return an Iterator object. (figure 3)

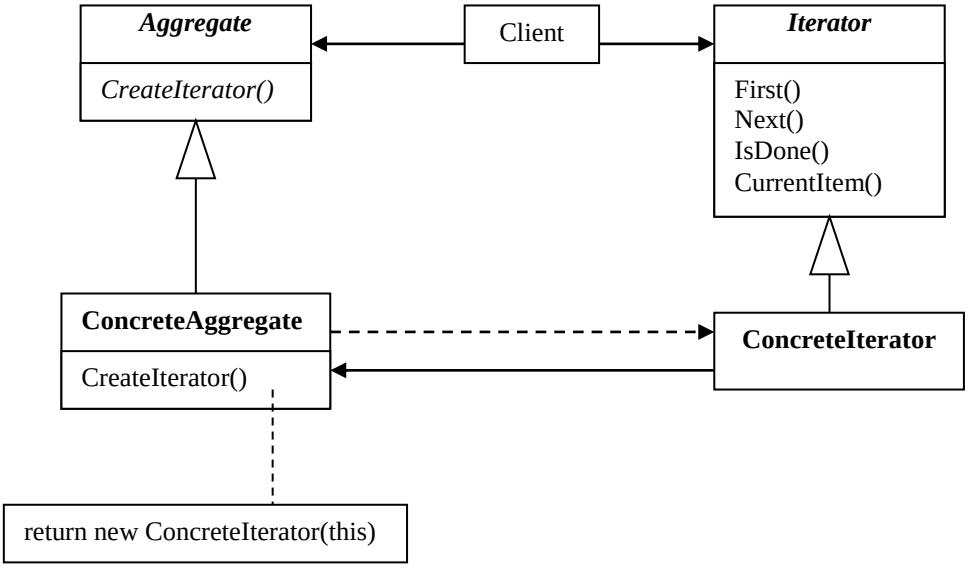


*Figure 3: The 'Iterator' pattern from Gamma et al (1995)*

In Java 1.1. the 'Enumeration' interface performs a similar role to the Iterator pattern, though with a smaller set of methods:

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

Container classes in the java.util package have an 'elements' method that returns an Enumeration object, for example this method from the 'Vector' class:

```
public class Vector extends Object
{
…
     public final synchronized Enumeration elements();
…
}
```

It should be noted that in Java 2.0, 'Enumeration' is renamed 'Iterator', its 'nextElement' method has been renamed 'next' and 'hasMoreElements' is renamed 'hasNext'. However it also has a less consistent interface with the addition of a 'remove' method that removes from the collection the element most recently returned by the iterator.

Given the Iterator pattern and the Enumeration (from Java 1.1) interface we can see that they differ in that the Enumeration has no equivalent of the 'First' method and that 'Next' and 'CurrentItem' are combined into a single 'nextElement' method. This means that we cannot use the same enumeration to traverse a container more than once, and that we cannot return the same object more than once in a single traversal. In a later example, we will see how we can reconcile these differences using another pattern.

Patterns in the AWT

The AWT (Abstract Windowing Toolkit) provides a number of examples of patterns in action, in particular the 'Composite' pattern that allows various types of GUI component to be nested inside each other. There is also a variation on Observer that allows 'Listeners' to be attached to components and a number of more complex patterns embedded in the implementation such as 'Abstract Factory' and 'Bridge' (that allow for a standard GUI system to be implemented on multiple platforms). The layout managers are examples of the 'Strategy' pattern, which is described later and enables a single frame to dynamically apply different layouts for it components, rather than different subclasses of Frame being used with fixed layouts. The other patterns are not detailed here, but the AWT provides a useful context for demonstrating a range of design patterns, as can be seen 'Principles of Object-Oriented Programming in Java 1.1' (Cooper, 1997) which implements eleven Gamma patterns using AWT examples. The relationship between patterns and the AWT is documented by Gamma and Johnson on a patterns home page (see references).

Reflection

One of the APIs in Java implements 'Reflection' which is not so much a design pattern as a 'pattern language', i.e. it is a generic term that is used to describe all kinds of systems where a 'meta level' is used to allow programs to change their behaviour dynamically. The Java Reflection API is a very constrained form of reflection known as 'introspection' which allows the internal state and structure of an object to be examined. The more general aspects of reflection are described by Buschmann (1996).

## Section 2: Instantiating Patterns in Java

Having recognized that some patterns are already evident in Java, the next step in using patterns in the curriculum is to code applications by applying patterns wherever appropriate. In this section we begin by referring back to the Enumeration class and the Iterator pattern and see how we might reconcile their differences by applying the 'Object Adapter' pattern. Then we look at a very simplistic scenario based around an electronic security system. In different aspects of this system we see how we might be able to apply patterns such as Façade, State, Strategy and Null Object.

From Enumeration to Iterator via Adapter

We have seen that the 'Iterator' pattern and the 'Enumeration' interface have some common aspects but also some major differences. We might feel that the larger Iterator interface is more appropriate for our needs than the two Enumeration methods, so how can we reconcile the two? One pattern that can provide a solution is the 'Adapter', which comes in two versions, the object adapter and the class adapter. Since the class adapter relies on multiple inheritance, we will use the object adapter here. The object adapter pattern in shown in figure 4. Its role is to convert an existing class interface to one that clients expect.

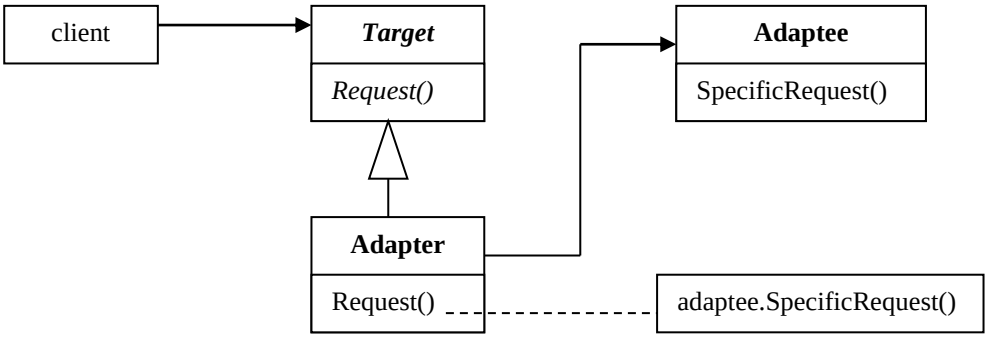


*Figure 4: The 'Object Adapter' pattern from Gamma et al (1995)*

We could apply the Adapter pattern to convert the existing interface of the Enumeration class to an interface that matches the Iterator pattern.. We can easily map the Enumeration and Iterator classes onto the object adapter pattern in figure four. The Enumeration would be the 'Adaptee' that would eventually receive the client request, but this would first be adapted by the concrete iterator appropriate to a given aggregate (container). This 'Adapter' would implement the interface of the 'Target', namely our abstract Iterator interface. This is, however, only half the story. To fully provide an implementation of the Iterator pattern we would also have to provide an 'Aggregate' by wrapping the various Java container with their own object adapters. These adapters would then be able to implement the 'CreateIterator' method.

Facades and the Law of Demeter ('Don't Talk to Strangers')

Figure 5 shows some classes that might be included in software for a security system. We might envisage a number of other components in such a system as well as the 'Camera', 'Light'. 'Sensor', and 'Alarm' classes included here.

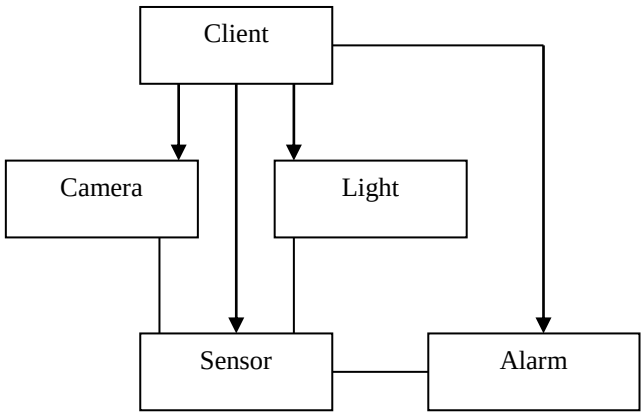


*Figure 5: Complex lines of communication where the various interface of a subsystem are exposed to client code*

Writing a system that uses these components might result in complex lines of communication if a client has direct links to most or all of these objects, in addition to the associations within the subsystem itself. The client would also have to know all the different component interfaces. The Façade pattern is a simple technique that involves routing communication to the various classes in a given sub system via a

single class. This loosens the coupling between client code and the specific subsystem classes and simplifies the use of the subsystem's functionality (figure 6).

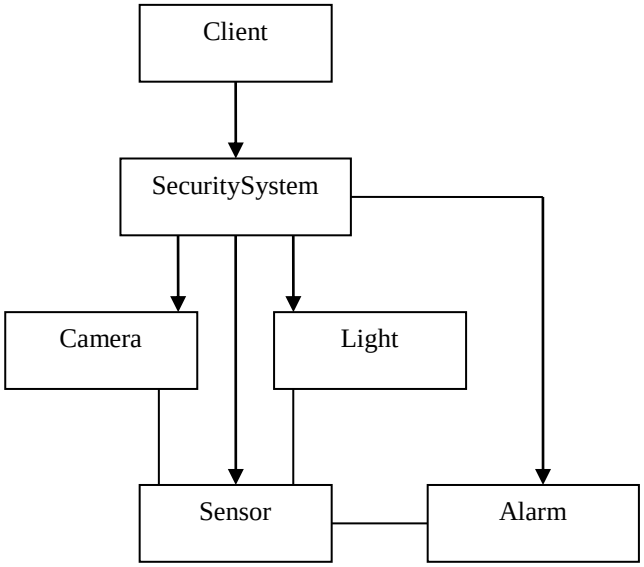


*Figure 6: The SecuritySystem class acting as a Façade to the classes in the underlying subsystem*

By introducing a Facade between the client and the subsystem classes, we provide a single point of contact. A basic type of facade returns references to objects from within the subsystem for example the public interface to this system:

```
public class SecuritySystem
{
// the constructor creates and populates the arrays
// using array sizes passed as parameters
   public SecuritySystem(int camera_count, int light_count)
      {…}
// return a camera object using an index number to locate it
   public CCTVCamera getCamera(int number)
   {…}
// return a light object using an index number to locate it
   public SecurityLight getLight(int number)
   {…}
}
```

In this instance, the client still has to be aware of the interfaces of both the 'Camera' and 'Light' classes. Client code will include statements like this:

```
system.getCamera(1).turnOn();
system.getCamera(1).showCamera();
system.getLight(1).turnOnManual();
system.getLight(1).showLight();
system.getLight(3).turnOnAuto();
system.getLight(3).showLight();
```

A more appropriate way of writing a facade is to apply the 'Don't Talk to Strangers' pattern (Larman, 1998) also known as the 'Law of Demeter' (though it should be noted that Larman's pattern encompasses only its simplest aspects). This 'law' says that a client should not need to have knowledge of the interfaces of any objects other

than its immediate collaborators. Therefore the security system client should only need to know about the interface of the Facade, not the objects behind it. The methods of the Facade should not return references to other objects.

A façade that provides similar functionality to the direct Light and Camera associations might look something like this:

```
public class SecuritySystemFacade
{
     public SecuritySystemFacade(int camera_count, int light_count)
     {…}
     public void turnOnCamera(int number)
     {…}
     public void turnOffCamera(int number)
     {…}
     public void showCamera(int number)
     {…}
     public void turnLightOnManual(int number)
     {…}
     public void turnLightOnAuto(int number)
     {…}
     public void turnLightOff(int number)
     {…}
     public void showLight(int number)
     {…}
}
```

One potential problem of attempting to follow the Law of Demeter in this way is that it leads to a 'fat' interface. Instead of a simple façade with a few methods to return objects, we end up with a much larger interface that may well have the same number of methods as the sum of all methods on all subsystem classes. However we can encapsulate a lot of complexity behind such a façade and perhaps reduce the number of methods if not all subsystem object methods need to be accessible by the client.

The State pattern – changing behaviour without selection statements

One of the implicit assumptions of object-oriented programming is 'selection statements considered harmful'. The key motivation for using inheritance and polymorphism is so that we can decouple our control structures from the type of entities that they manipulate. Thus we can replace this kind of (pseudo) code:

```
switch (drawing_type)
{
     case circle :
          drawCircle();
          break;
     case rectangle:
          drawRectangle();
          break;
// etc..
```

with this

```
shape.draw();
```

assuming that 'shape' is a superclass reference to a subclass object and that 'draw' is a

polymorphic method.

However, we often find that our code is still littered with conditional statements due not to object type but to object state. This tends to be particularly evident in event driven code where certain methods (e.g. 'paint') are invoked automatically and we have to work out what to do each time the method is called depending on the current state of the application. Returning to our security system example, we might respond differently to a 'show light' message depending on the state of the light. It may be off, on permanently or on but switched automatically (i.e. linked to a sensor). Therefore our method consists of a switch statement:

```
public void showLight()
{
   switch(light_state)
   {
      case ON_MANUAL :
         System.out.println("light " + id_number + " ON (manual)");
         break;
      case ON_AUTO :
         System.out.println("light " + id_number + " ON (auto)");
         break;
      case OFF :
         System.out.println("light " + id_number + " OFF");
         break;
      default :
         System.out.println("state undefined");
   }
}
```

Even for a trivial example such as this we can see the lack of flexibility in the code. We can improve this antipattern by applying the 'State' pattern (Gamma et al 1995). This pattern allows us to change the behaviour of an object according to its state by using state objects rather than conditional code (figure 7)

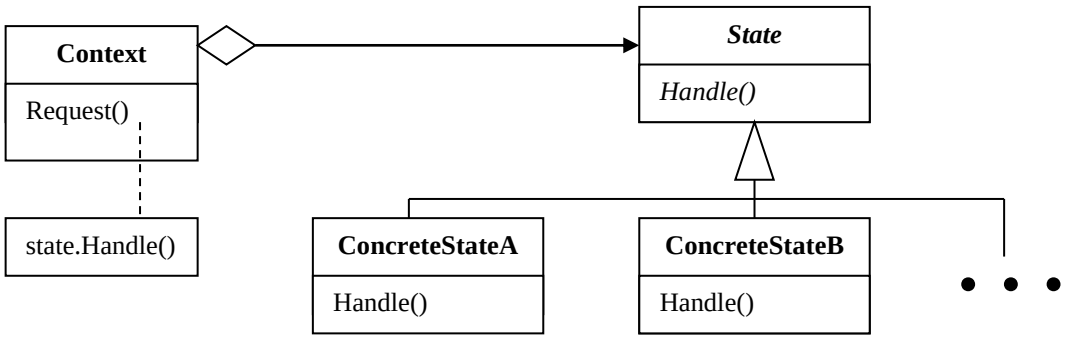


*Figure 7: The 'State' pattern from Gamma et al (1995)*

We could for example use State objects to remove the 'switch' statement from the showLight method of the Light class. If we provide an abstract 'LightState' class with subclasses 'LightOffState', 'LightOnManualState' and 'LightOnAutoState' then we can swap between these states, delegating the 'showLight' method to the state object:

```
public class SecurityLightS
{
```

```
     private int id_number;
     private LightState state;
     public SecurityLightS(int number)
     {
          id_number = number;
          state = new LightOffState(id_number);
     }
// methods that change the state
     public void turnOnManual()
     {
          state = new LightOnManualState(id_number);
     }
     public void turnOnAuto()
     {
          state = new LightOnAutoState(id_number);
     }
     public void turnOff()
     {
          state = new LightOffState(id_number);
     }
// 'showLight' delegates its behaviour to its current state object
     public void showLight()
     {
          state.showLight();
     }
}
```

Although this example may look unnecessarily complex there are several important aspects to consider. First, we are only applying a single method to the state object, but there might be many, allowing us to remove many switch statement from various methods of the security light class. Second, the methods that switch state may well be event driven, in which case they would need to exist anyway to respond, for example, to GUI events. Rather than simply changing the values of flag variables, we can use these methods to directly change state. Another important aspect is that this pattern allows us to add new state subclasses without affecting the client very much. All that is required is that the new state is created in response to the appropriate event. Finally, if the class has a well-defined set of states and transitions, we might directly realise a state diagram in the implementation of the state objects so that they can trigger each other rather than the enclosing object having to explicitly change from one state to another.

Strategy and Null Object patterns

The state pattern is similar to two other patterns, ‘Strategy’ and ‘Null Object’, that use polymorphism within the implementation of a class to reduce the need for conditional statements. Whereas State is used to replace conditional statements based on object state, Strategy is used to replace conditional statements based on object behavior such as algorithms (figure 8).

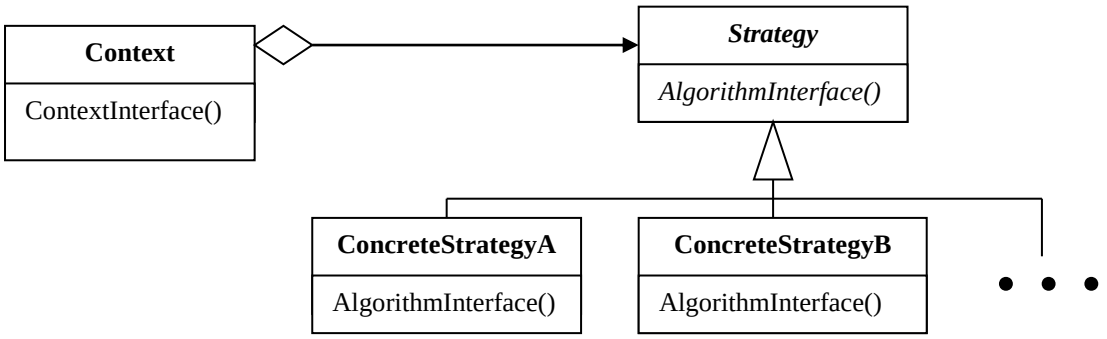


*Figure 8: The 'Strategy' pattern from Gamma et al (1995)*

Where we have a choice of algorithm to achieve the same ends, we can replace conditional statements with objects that encapsulate the various algorithm implementations. For example games that rely on searches to predict future moves could switch between (perhaps) depth first and breadth first searches to provide different levels of play. An example where we might apply the strategy pattern in the security system might be to allow for different camera panning strategies within a single SecurityCamera class. Rather than creating a hierarchy of camera objects, each with a different panning algorithm, we can create a more flexible system where cameras can switch between strategies. Like the State pattern this also makes it easier to add new subclasses of strategy without major changes to the client (SecurityCamera).

Null Object is a similar pattern to both State and Strategy, in this case removing the necessity to check for 'null' references in code by using a Null Object that has the same interface as a 'real object' but responds by doing 'nothing' in the appropriate way (figure 9).

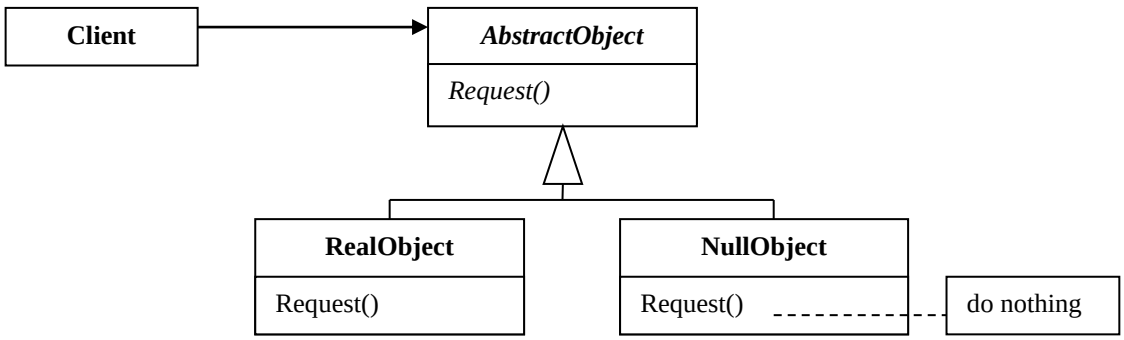


*Figure 9: The 'Null Object' pattern from Woolf (1997)*

The Null Object pattern allows us to remove conditions from code such as:

```
if(camera1 != null)
{
     camera1.turnOn();
     //…
```

or similarly code that uses 'try..catch' blocks to catch NullPointerExceptions in similar circumstances. If 'camera1' is not available to be turned on, rather than using

a null reference that cannot receive messages, we replace it with the null camera. Null Object can pose some problems in replacing references to 'null', for example we are no longer able to apply 'if…then…else' conditions to the references. Whether this makes null object inappropriate in a given context depends on how much we want to do when we encounter null objects in different circumstances. Sometimes we do not really want an object that does 'nothing', but one that does something different from 'real' objects.

## Assignment work and patterns

Even at the level of a small-scale student assignment, it is not difficult to apply some of the simpler patterns to make a better architecture and to encourage students to think about design issues. The security system example used here is intended purely as a simple context in which to describe patterns, but its emphasis on hardware devices and potential difficulties in scaling up to a realistic but still feasible task means that it would not be an ideal assignment. However the nature of patterns as generally applicable micro architectures suggests that there can be few contexts where at least one or two patterns could not be usefully applied.

## Summary

This tutorial has focussed on a few of the simpler design patterns and sought to use Java as both a context and a tool for patterns learning. Its intention was to demonstrate that the combination of Java and patterns could provide us with well-engineered object-oriented systems even at the modest scale of a student assignment. Thus within the computing curriculum, Java plus patterns can indeed equal object engineering.

## References


Brown, W., Malveau, R., McCormick, H. and Mowbray, T. *Anti Patterns: refactoring software, architectures and projects in crisis*. New York: Wiley 1998.

Buschmann, F. *Reflection.* Pattern Languages of Program Design 2, ed. Vlissides, J., Coplien, J. and Kerth, N. Reading, Mass: Addison-Wesley 1996, p. 271-294.

Cooper, J. *Principles of Object-Oriented Programming in Java 1.1*. NC: Ventana, 1997.

Coplien, J. *Advanced C++ Programming Styles and Idioms*. Reading, Mass.: Addison-Wesley 1992.

Coplien, J. A Generative Development-Process Pattern Language. Pattern Languages of Program Design, ed. Coplien, J. and Schmidt, D. Reading, Mass: Addison-Wesley 1995, p. 183-237.

Flanagan, D. *Java in a Nutshell.* Cambridge: O'Reilly 1997.

Fowler, M. *Analysis Patterns*. Reading, Mass.:Addison-Wesley 1997.

Gamma, E. and Johnson, R. *JavaAWT*, (from Patterns home pages) http://st-

www.cs.uiuc.edu/cgi-bin/wikic/wikic?JavaAWT.

Gamma. E, Helm. R., Johnson. R. and Vlissides, M. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley 1995.

Grand, M. *Patterns in Java volume 1*. New York: Wiley 1998.

Larman, C. *Applying UML and Patterns*. NJ: Prentice Hall 1998.

Riehle, D. *Composite Design Patterns*. Proceedings of OOPSLA 97, ACM 1997.

Woolf, B. *Null Object*. Pattern Languages of Program Design 3, ed. Martin, R. Riehle, D. and Buschmann, F. Reading, Mass.: Addison-Wesley 1998.