# Java's Reflection Model and Design Patterns

Youssef Hassoun
King's College London
**35** PUBLICATIONS **554** CITATIONS
SEE PROFILE

Roger George Johnson
Birkbeck, University of London
**54** PUBLICATIONS **377** CITATIONS
SEE PROFILE

# Java's Reflection Model and Design Patterns

**Youssef Hassoun, Roger Johnson**
School of Computer Science and Information Systems
Birkbeck College, University of London, Malet Street, London WC1E 7HX, UK
{yhassoun, rgj}@dcs.bbk.ac.uk
**Steve Counsell**
Department of Information Systems and Computing
Brunel University, Uxbridge, Middlesex UB8 3PH, UK
{steve.counsell}@brunel.ac.uk

## Abstract

Dynamic proxies support meta-programming and extend the reflective abilities of Java beyond pure introspection. Proxy objects are interpreted as meta-objects that control the behaviour of application objects at the base level. In this paper, we investigate the implications of applying reflective programming techniques provided by proxies. We address the problem of how such proxies can support open implementation for customising application behaviour. We then examine the consequences of adapting design patterns to a proxy-based reflection model. Our research shows dynamic proxies to be a valuable mechanism for understanding Java language characteristics in greater detail and also an aid to object-oriented reuse.

## 1. Introduction

Reflection or meta-programming as it is sometimes known is the ability of a computational system to reason about and act upon itself [24, 32]. A reflective system incorporates data representing static and dynamic aspects of the system itself. This self-representation makes it possible for the system to manipulate its own state. Java's reflection model supports introspection where a program can observe and reason about its own state. The model also supports behavioural reflection, defined as the ability to intercept an operation such as method invocation and alter the behaviour of that operation [11].

Reflective systems are characterized by a multi-level architecture separating meta-levels from the base level [24, 32]. In reflective object-oriented systems, certain aspects of the base level are represented or *reified* as *meta-objects* to provide a representation of the execution state as data. The role of meta-objects is to observe and modify the base objects they represent (their referents). The code dealing with meta-objects is called a meta-program and its interfaces are called meta-object protocols. Application objects populate the base level and the program on which reflective computation is performed is called the *base program*. Compared to static systems exhibiting the same interface, reflective systems show a lower level of *object coupling* and consequently are more reusable and (potentially) easier to maintain [13, 14, 15].

Java's dynamic proxies provide a mechanism allowing us to intercept operations on objects and modify their behaviour at runtime [11]. A proxy's invocation handler can be interpreted as a meta-object residing in the meta-level of a reflective architecture where application objects populate the base level. Using reflection, the internal implementation of a base program can be accessed and the behaviour of base objects customised. Reflection and the interception mechanism of dynamic proxies allow us to customise application behaviour without modifying the application code or the underlying Java Virtual Machine (JVM). The customisation code is implemented as a meta-program. Meta-objects represent generic behaviour independent of applications and can be reified by instantiating corresponding proxies.

The customisation code implemented at the meta-level corresponds to the optional meta-interface of the open implementation approach for developing maintainable software systems [21]. An open system permits its clients to take control of implementation issues of system components allowing clients to customise the components according to their needs. Open architectures (following the open implementation design) are characterized by separating applications from an optional meta-interface. The base program provides basic functionality (base level interface) and basic structure (default implementation) and remains executable without the meta-program. The meta-program allows the user to add customisations from above (the meta-level).

In this paper, we review the reflection model based on dynamic proxies and provide some examples showing how it can support open implementation. We also address the question of adapting design patterns catalogued in the book of Gamma et al. [8] to the model. Although many patterns make use of inheritance, Gamma et al. propose programming to interfaces and favour object composition over class inheritance as guiding principles for pattern implementation. The proposed reflection model provides a mechanism based on flexible runtime object composition; separation of base level applications and meta-programs promotes loose coupling and code reuse.

The key contribution of this research is that by adopting a reflection model supported by Java's dynamic proxies, applications can be customised and adapted to new requirements without the need to change their default behaviour and, hence, at relatively low cost. In addition to applications running on a JVM, our approach is also applicable to client/server applications running on several JVMs distributed over a TCP/IP network. Moreover, we show how and whether design patterns can be adapted to the proposed model and discuss the impact in terms of simplifying pattern implementation, coupling, code reuse and modularity. The research described thus reviews the meta-level approach adopted in [11, 12] and applies the principles to design patterns.

The paper is arranged as follows. The next section deals with related work. In Section 3, the Java reflective API, including the introspective interface and behavioural reflection supported by the proxies, is described. In Section 4, we show how to apply the reflection model to customise application behaviour by following the open implementation approach. Section 5 addresses adapting design patterns to the proposed reflection model. In Section 6, various features of the reflection model are discussed and finally, in Section 7, we draw conclusions and discuss future work.

## 2. Related work

Our reflection model supports the principle of open implementation [21]. The goal of open implementation is to give clients some control over the implementation strategy of the modules while retaining the advantages of the traditional closed implementation modules [23]. Our goal is to allow clients to modify the behaviour of their applications by providing them with reusable code and a meta-interface to access this code. In our model, base applications provide default behaviour and remain executable without the binding to the meta-interface. This corresponds to the default implementation of the clients' module. In both cases, the meta-interface provides access to customise the default implementation.

Kiczales et al. [20] proposed meta-object protocols (MOPs) as a technique to incrementally modify the behaviour and the implementation of the Common Lisp Object System (CLOS) [4]. Here, the underlying programming language becomes the application domain and the borderline between the users of the language and its designers disappears. In our work, applying the reflection model is restricted to customising Java *applications* and we leave the question of applying the model to the Java language itself (including its libraries) for future research.

Separating applications from their meta-level representations has parallels with the aspect-oriented programming (AOP) approach [22]. Most AOP languages are extensions of existing OO languages [2] and provide additional mechanisms and constructs supporting the separation of concerns. In past research, reflection capabilities of the underlying OO languages have been used to allow for runtime weaving and to implement reusable code. For example, in [5], a hybrid system was proposed where join points are defined at compile-time (like AspectJ), but aspects are weaved at runtime, using reflection. In [10], a general approach was proposed for merging generic aspects with application code. The aspects were implemented as self-contained Hyper/J hyperslices and merging was realised at compile-time using the corresponding scripting language and tool. In [9], Hannemann and Kiczales studied the crosscutting structure of design patterns and provided pattern implementations in AspectJ, showing better modularity as a result. Marcos et al. [25] proposed a reflective two-level model in which design patterns were represented as meta-objects and provided implementations of some patterns in CLOS. We approach the problem of applying reflective techniques to design patterns from a different perspective; we do not use AOP language extensions and restrict ourselves to the behavioural reflection model supported by Java's dynamic proxies. We show that applying this model lessens coupling, promotes code reuse and, in some cases, makes the pattern implementation less complex.

## 3. Java reflective API

With JDK 1.1, Java included an API for accessing meta-objects. The core reflection package consists of the classes of the `java.lang.reflect` as well as the class `java.lang.Class`. This API represents a reflective model that allows introspective inspection of a system's structure as well as behavioural reflection.

### 3.1. The introspective interface

Introspective structural reflection is the ability of a program to obtain information about its structure including its classes, its hierarchy and the instantiation relationships of its objects. Java's structural reflection model is supported by `Class` and associated classes like `Method`, `Field` and `Constructor` of the `java.lang.reflect` package. `Class` is of particular interest because each object of an arbitrary system has an associated unique instance of this class representing its meta-object. By invoking the method `getClass()` on application object (base_`obj`), we get its *reified* class. The method is part of the `java.lang.Object` public interface.

In reflective models, *reification* is an essential concept. Reification means making certain aspects of the lower level explicit so that these aspects can be inspected and manipulated by applications. The aspects of a system's base level that can be reified depend on the reflective model. In Java's structural reflective model, the reified entities allow the inspection of the *structure* of the system application; i.e., its classes, their methods, fields, constructors, interfaces and super-classes.

In Java, the `Object` class lies at the root of every inheritance hierarchy and, consequently, the class of any application object can be reified and used to expose the object's structure. A complete description of the *type*[1] of a base object may be obtained by invoking the various methods of `Class` on the corresponding meta-object. The methods allow us to obtain representations of the class's constructors, methods, fields, super-class and interfaces. These

[1] In an OO system, a type summarizes the common features of a set of objects with the same characteristics. In conventional OO languages, in addition to built-in types, user-defined classes imply or comprise type definitions.

representations in turn provide methods that allow their structure, e.g., the types of fields and the parameter types of methods, to be discovered. The process can be continued recursively to cover the entire class hierarchy of the object under consideration.

`Class` is *final* and has no public constructors meaning that it cannot be extended by inheritance and that it cannot be instantiated. Instead, `Class` objects are constructed automatically by the JVM as classes are loaded and by calls to the `defineClass()` method in the class loader. Similarly, the classes `Method`, `Field` and `Constructor` are also defined as final and provide no public constructors. This shows that introspective reflection is limited to obtaining information about objects' structures. In addition, the reflection model supported by these classes is fixed by the Java system and cannot be changed. Accordingly, the class of an application object can be reified at runtime and the object's structure can be uncovered. It is not possible to change either the behaviour of this object or its structure, however.

**3.2. Dynamic proxies and behavioural reflection**

A proxy is a program that provides a communication bridge allowing different applications (clients and targets) to engage and exchange data. From a design perspective, a proxy is a design pattern [8]; it describes a general solution to common design problems that occur repeatedly in different contexts. In distributed systems, including web applications, a proxy is a service that sits between application servers and clients. This service receives requests from clients and makes requests to servers on behalf of the clients.

In addition to *normal* proxies, Java supports dynamic proxies. A Dynamic Proxy API was introduced in Java 2 Standard Edition (J2SE) version 1.3. Normal proxy classes are usually available as byte-codes after having been compiled from the corresponding Java source files. The byte-codes are loaded into the JVM before proxy objects are instantiated. The byte-codes of Java's dynamic proxies, on the other hand, are generated at runtime. There are advantages to using dynamic proxies instead of *normal* proxies, particularly in relation to design and consequently program maintenance. Firstly, from an Object-oriented design (OOD) point of view, dynamic proxy classes exhibit less coupling to target classes when compared to the static case with normal proxies. There is no direct relationship between these two class groups; the coupling takes place dynamically at the object level. In fact, the type (set of implemented interfaces) of a dynamic proxy is set to that of the target object upon its creation at runtime. The type of a normal proxy, on the other hand, is set at compile-time. Secondly, the dispatching mechanism of dynamic proxies allows, using polymorphism and reflection, for implementing generic behaviour independent of the targets' type.

With the introduction of dynamic proxies, the reflective power of the Java language has been extended to allow for behavioural reflection. A proxy instance dispatches method invocations to an associated invocation handler before these invocations are further delegated to the corresponding target object(s) (Figure 1). We can use dynamic proxies to intercept method invocations on base objects to modify or customise the behaviour of these objects by executing reflective code implemented in the invocation handler.

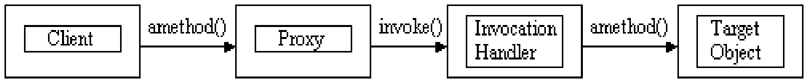

**Figure 1. Method invocations are dispatched to InvocationHandler**

A behavioural reflection model allows programs to intervene in the current execution in order to execute some reflective code analyzing or modifying the course of events [6]. There are different implementations of the model depending on the programming language. Java supports behavioural reflection through dynamic proxies.

In [7], Ferber classifies different models of reflection according to the relationship between meta-entities and base entities. We follow a reflective model based on the meta-object approach. In this model, each base-object has a potential meta-object associated with it and the meta-object representation is restricted to controlling the behaviour of referents (and does not include their states). In [11], a two-level reflective architecture supporting the meta-object approach to reflection was proposed. Figure 2 depicts the reflective architecture where proxy objects at the meta-level include invocation handler associated with proxies created at runtime. The proposal was based on:

1) the interpretation of invocation handler of dynamic proxies as meta-objects,
2) the causal relationship between meta-level and base level is defined in terms of mechanisms associating a base object to an invocation handler, and
3) the events that trigger the meta-level being method calls on proxy objects. The events trigger the reification process in which a meta-object modifies the *behaviour* of its referent. In return, any changes made at the meta-level are reflected back on the reified elements at the base level.

Upon instantiating a proxy, a base object is associated with the meta-object thus reified. The proxy object acquires the type(s) of the base object without implementing its interfaces explicitly.

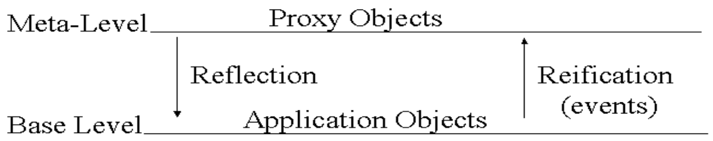

**Figure 2. A two-level reflective architecture with proxies**

## 4. Open implementation and dynamic proxies

In computer science, the principle of abstraction plays a fundamental role in overcoming the complexity inherent in large software systems. Abstraction is used as a means by which uninteresting details can be removed while keeping desired essence.

The traditional way of applying abstraction follows the principle of information hiding according to which implementation details are separated from and hidden behind a module's *interface*. The interface represents an abstraction *barrier* separating clients from implementation. Data provided at the abstraction barrier is the only means by which clients can manipulate the abstraction; this strategy of information hiding is also known as black-box abstraction.

The concept of information hiding was first introduced by Parnas [31]. He argued that in implementing a system, difficult design decisions or decisions which are likely to change should be assigned to modules and not to subroutines. Design decisions transcend execution time and therefore design modules may not correspond to processing steps. Each module should be designed so as to hide the internal details of its processing activities and modules communicate only through well-defined interfaces. According to Meyer [26], information hiding emphasizes separation of a module's function from its implementation. Meyer defines information hiding as a *selected subset* of the module's properties that can be made available to the authors of client modules. A principle related to information hiding is that of encapsulation, which is part of the OO paradigm and supported by OO languages. Encapsulation is sometimes used as a synonym for information hiding, but the two are different. Encapsulation is closely related to the concept of Abstract Data Type (ADT) which bundles data with methods that operate on that data. Often data abstraction as implemented in

ADT is wrongly interpreted to mean that data is somehow hidden. Encapsulating data does not necessarily imply hiding data, for we can, for example, define a class (OO equivalent to ADT) where data members are directly accessible.

Despite the benefits of applying black-box abstractions such as loose coupling and module reuse, there are cases where it is advantageous to open implementations and allow clients to change them [21]. In cases where default behaviour is inappropriate, clients can customise an implementation according to their needs *without* changing its default behaviour. The underlying architecture is a two-level reflective architecture in which the default implementation resides at the base level. The meta-level is optional and can be used if the default implementation is inadequate to meet client's requirements. The MOPs approach is an example of a programming technique used to realise the open implementation design principle in CLOS.

The reflection model proposed in Section 3 provides a means of realising the open implementation design principle. We can use the model to customise arbitrary applications without changing their default behaviour. There are two alternatives to achieve this goal. The first is by following the MOPs approach and customising the underlying programming language (in our case Java). The second alternative is by providing a set of meta-objects implementing generic behaviour as a library. Meta-objects can be used to change applications' behaviour at runtime. In this paper, we follow the second alternative. We consider examples showing how generic behaviour can be implemented and used to customise arbitrary applications using the reflection model.

In the following example, we note that the base level application remains executable without being attached to the meta-level. The meta-level program is optional and can be used to customise particular aspects of the application to better meet new requirements.

## 4.1. Customising applications' behaviour

Consider a tracing meta-object, which intercepts all method invocations of application objects it represents such that method related information before and after the method is called are accessible. The tracing code implemented at the meta-level using the proxies can be made available to any application as part of a debugging and testing framework, without the need to change the application code. We observe that the tracing strategy can be modified without affecting potential client applications; applications can use the code without the need to change the behaviour or the structure of their objects.

Creating a proxy constitutes the first step in the process of customising application behaviour. Figure 3 shows the class `ProxyFactory` responsible for creating proxies. The base object is passed as a parameter to the invocation handler (here `TracingHandler`). The returned proxy object acquires the types (interfaces) of the base-object. Invoking base methods on the proxy triggers the reification process in which the behaviour of the base-object can be modified. The reified meta-object of type `TracingHandler` inspects the structure of the base object and uses the collected data to pass debugging information about its referent.

We note that `ProxyFactory` does not represent a creational design pattern as the Factory patterns of [8]. With `createProxy(Object base_obj)`, base objects can register themselves and be attached to meta-objects at the higher level. The returned proxy object can be used to represent base objects. The class depends on the type of the invocation handler and the proxy classes of `java.lang.reflect` package but is independent of application. The same applies to the invocation handler class (here `TracingHandler`–Figure 4) implementing the customisation code.

```
import java.lang.reflect.*;

public class ProxyFactory {
  public Object createProxy(Object base_obj) throws Throwable {
    Class c= base_obj.getClass();
    ClassLoader cl=c.getClassLoader();
    Class[] infs=c.getInterfaces();
    InvocationHandler inh=new TracingHandler(base_obj);
    Object proxy = Proxy.newProxyInstance(cl, infs, inh);
    return proxy;
  }
}
```

**Figure 3. ProxyFactory for creating proxy objects (proxies and handler)**

```
import java.lang.reflect.*;

public class TracingHandler implements InvocationHandler {
  private Object base_obj = null;

  public TracingHandler(Object base_obj) {
    this.base_obj = base_obj;
  }
  public Object invoke(Object proxy, Method m, Object[] args) throws Throwable {
    Object result = null;
    String cname=base_obj.getClass().getName();
    try {
      System.out.println(cname + "." + m.getName() + " will be invoked.");
      result = m.invoke(base_obj, args);
    } catch(Exception e) {
     System.out.println(cname+"."+m.getName()+"caused an exception "+e.getMessage());
    } finally {
      String ret_val=(result == null ? "null" : result.toString());
      System.out.println(cname+"."+m.getName()+"- invocation complete, "+ret_val);
    }
    return result;
  }
}
```

**Figure 4. Generic TracingHandler for intercepting method calls on application objects**

The tracing example depicted in Figures 3 and 4 shows that it is possible to implement generic reusable code to trace arbitrary applications. In this way, the behaviour of an application is customised without changing application code. To use the generic meta-program, proxy objects should be created through which application objects can be attached to invocation handler (acting as meta-object). Figure 5 contains pseudo-code showing how tracing of base objects can be *reified* by creating a proxy, passing application objects and then invoking the methods on the proxy.

```
public class TracingDemo {
  public static void main(String [] args) throws Throwable {
    ProxyFactory pf=new ProxyFactory();
    // Here, <Base Object> is of an instance of <Application Type>
    // <Application Type> interface includes a number of methods: m1, m2 etc.
    <Application Type> iat=(<Application Type>)pf.createProxy(<Application Object>);
   <Base Object>.m1(<parameter list>);
   <Base Object>.m2(<parameter list>);
   // etc…
  }
}
```

**Figure 5. Generic TracingHandler for intercepting method calls on application objects**

## 4.2. Customising distributed applications

The example described in the forgoing section shows how the proposed behavioural reflection model supports open implementation for applications running on a single JVM. The model can also be applied to a distributed environment where the base level and the meta-level need not be part of the same environment and can be made to communicate over different protocols supported by the Java language. The most important protocols are the JRMP (Java Remote Method Protocol or the RMI Protocol), HTTP (HyperText Transfer Protocol), IIOP (Internet Inter-ORB Protocol) or a combination thereof. Java provides a number of network programming facilities to allow the development of distributed applications. Lower level networking such as communication with sockets, Web URLs, and datagrams are supported. In addition, Java supports higher-level object-oriented networking including communication with homogenous RMI (Remote Method Invocation) [19] and heterogeneous CORBA (Common Object Request Broker Architecture) of the Object Management Group (OMG) [18]. In addition, Java's Servlets API allows web developers to extend the functionality of a Web server and to access existing business systems [17].

In this section, we concentrate on object-oriented networking and explain some networking techniques by reconsidering the tracing example of the previous section and adapting it to an RMI-based system. In doing so, we differentiate between base and meta-objects and associate base objects with clients' applications and meta-objects with server applications. More examples on extending single JVM applications to support customisation of application behaviour in distributed systems based on the CORBA architecture or on the Java Servlets model can be found in [12].

Distributing a Java system over a network usually means splitting the system into different clients communicating with a server (or servers) running on different JVMs. Base objects belong to client applications whereas classes implementing generic behaviour (meta-objects) including classes responsible for delivering various proxy objects are part of server applications.

The first step in constructing the RMI version of the tracing application is to transform the class responsible for creating a proxy `i`nto a *remote object*. A remote object implements a *remote interface* (Figure 6) and extends `UnicastRemoteObject` allowing unicast (point-to-point) remote communication by using RMI's default sockets-based transport for communication.

```
Public interface IProxyFactory extends java.rmi.Remote {
  public Object createProxy(Object base_obj) throws java.rmi.RemoteException;
}
```

**Figure 6. Remote interface defining the remote method implemented on the server-side**

Figure 7 shows the changes needed to transform `ProxyFactory` into a remote RMI object. The client communicates with the remote object on the server side by calling the method on a stub (a proxy), provided by the RMI system to represent the remote object. This call is forwarded to the next layer in the communication interface where RMI's invocation semantics is supported. The stubs and skeletons needed to support the client-server communication can be generated by the *rmic*-tool provided by Java's RMI-system.

```
import java.lang.reflect.*;
public class ProxyFactory extends java.rmi.server.UnicastRemoteObject
                                                implements IProxyFactory {
  public ProxyFactory() throws java.rmi.RemoteException { super(); }
  public Object createProxy(Object base_obj) throws java.rmi.RemoteException{
    // as in Figure 3
  }
}
```

**Figure 7. Remote interface defining the remote method implemented on the server-side**

The meta-progam implementation does not change. However, a server application is required to start the server(s) and to register the remote object under an arbitrary name. The client looks up the remote object by its name in the server's registry. The client process obtains a reference to the remote factory object through the same name used. The server process then waits for client requests running on different JVMs on the network.

On the client side, the changes are trivial. All base objects are made serializable to conform to the RMI mechanism of sending object data over the network. All objects exchanged between client and server as parameters or as return types must be serialized before they can be sent over the network. The remote JVM de-serializes the objects and makes them ready for use by the application that needs them on the server side.

As for the client application, the major difference when compared with the client of the single JVM application is that the client cannot instantiate `ProxyFactory` object, because it does not run on its local machine. Instead, it looks up the remote object reference by using the RMI Naming facility. The client uses the naming facility to look up the remote object by name in the remote host's registry by passing a URL-string. Once the remote object is identified, processing follows as per the single application case.

We have seen that customising applications can also be realised over the network. Clients only need to lookup the proxy objects, which allow them to modify the behaviour of their objects. The meta-objects reside on the server side and can be accessed through the proxy objects.

## 5. Design patterns and dynamic proxies

Design patterns describe general solutions to common design problems. Some design problems appear repeatedly in different contexts; the main motivation behind patterns is to provide abstract solutions applicable in all contexts. In a particular context, however, application-dependent information must be provided. Gamma et al. provide a systematic approach and a catalogue of twenty-three design patterns [8]. Research on design patterns in software work was originally inspired by C. Alexander’s work on the problems of urban architecture and design [1].

In the subsequent analysis, the twenty-three patterns of Gamma et al. are analysed from the perspective of the reflective architecture supported by Java’s dynamic proxies. The dispatch mechanism provided by the proxies makes the implementation of some patterns qualitatively simpler. In some cases, the separation of concerns supported by the model helps to decrease the coupling among patterns’ participants. This becomes clear from the class structures expressed as UML diagrams [30]. In some other cases however, employing the reflection model shows no genuine advantages when compared with the *standard* (non-reflective) implementation of the patterns. This is due to the limitations of the reflection model and/or to the structure and/or application of the pattern itself. The model does not allow structural changes at runtime. The application domain of some patterns, e.g., compiler construction of the Interpreter pattern, makes the employment of the reflections model inappropriate. In some cases, separating the participants of a pattern into base and meta-objects becomes unfavourable when the structure of the pattern requires structural reflection. For some patterns, the Java language provides appropriate abstractions, e.g., Iterator and Prototype.

For several patterns we provide code extracts as and show the class structure before and after adapting the pattern to the reflection model. To keep diagrams simple, we show the relevant constituents of a structure and refrain from presenting details in the case where the content is evident. The idea is to highlight the resulting class structure of each of the patterns after adaptation to the refection model.

**The Template Method Pattern**

The Template Method Pattern defines the skeleton of an algorithm and defers some steps to sub-classes without changing the algorithm's structure. The pattern lies at the basis of object orientation where inheritance is used to extend and/or modify existing behaviour.

The Template Method pattern serves as a good example to show how dynamic proxies dispatching mechanism can be used as a substitute for inheritance. The "abstract" part of the Template Method algorithm implemented by super-class(es) and invoked by looking up the methods along the inheritance path. Using our reflection model, this part is moved to the meta-level and invoked through the dispatching mechanism supported by dynamic proxies.

```
import java.lang.reflect.*;                                                  //1
                                                                             //2
public class ProxyFactory {                                                  //3
  public Object createProxy(Object base_obj) throws Throwable {              //4
    Class c=base_obj.getClass();
    ClassLoader cl=c.getClassLoader();
    Class[] infs=c.getInterfaces();
    InvocationHandler inh=new TemplateMethodHandler(base_obj);
    Object proxy = Proxy.newProxyInstance(cl, infs, inh);
    return proxy;
  }
}                                                                             //12
class TemplateMethodHandler implements InvocationHandler {
  private Object base_obj;
  public TemplateMethodHandler(Object base_obj) { this.base_obj=base_obj; }
  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    try { this.perform(); } catch (Exception e) { e.printStackTrace();}
    return null;
  }
                                                                              //20
  public void perform() {                                                     //21
      firstStep(); ((ITemplateMethod)base_obj).secondStep(); thirdStep();     //22
  }
  private void firstStep() { System.out.println( "Metaobject.... firstStep" ); } //24
  private void thirdStep() {                                                  //25
    thirdStep_1();
    ((ITemplateMethod)base_obj).thirdStep_2();
    thirdStep_3();
  }
  private void thirdStep_1() {System.out.println("Metaobject.... thirdStep_1" );}//30
  private void thirdStep_3() {System.out.println("Metaobject.... thirdStep_3");} //31
}                                                                             //32
```

**Figure 8. An implementation of the Template pattern: the meta-program**

```
interface ITemplateMethod {                                                  //1
  public void secondStep();                                                  //2
  public void thirdStep_2();                                                 //3
  public void findSolution();                                                //4
}                                                                            //5
class BaseImpl implements ITemplateMethod {                                  //6
  public void secondStep() { System.out.println ("BaseImpl....   secondStep" ); }
  public void thirdStep_2() { System.out.println( "BaseImpl....   thirdStep_2" ); }
  public void findSolution() {}
}                                                                             //10
public class TemplateMethodDemoDynamic {
  public static void main( String[] args) throws Throwable {
    ProxyFactory pfac=new ProxyFactory();
    ITemplateMethod algo=(ITemplateMethod)pfac.createProxy(new BaseImpl());
    algo.findSolution();
  }
}
```

**Figure 9. An implementation of the Template pattern: the client part**

Figures 8 and 9 show an implementation of the Template pattern using dynamic proxies. The algorithm consists of three different steps defined at the meta-level (Figure 8, line 22). Some of the algorithm's methods are defined at the base level; `BaseImpl` implements the second step and part of the third step (Figure 9, lines 6-10). An implementation using inheritance replaces the proxy classes (`TemplateMethodHandler` and `ProxyFactory`) with super-classes (to `BaseImpl`) implementing the methods of the algorithm, namely, `firstStep()` and parts of the `thirdStep()` not implemented in `BaseImpl`.

**The Strategy Pattern**

The Strategy pattern makes a family of encapsulated and related algorithms interchangeable by separating the algorithms from the context in which they are used. Strategy is an example of how the proposed reflection model simplifies the implementation of the pattern. Figure 10 depicts the class structure of the Strategy pattern in UML notation. We note the dependency of the context represented by `StrategyContext` on the algorithms represented by `Strategy`.

Different applications provide different families of related algorithms. In other words, strategies are application dependent. In the Strategy pattern, the role of a context object is to control the execution of algorithms and these can be altered without any reference to the context. It follows that strategies can be assigned to the base level, because they are application dependent, and contexts can be lifted to the meta-level, because their *role* is general and independent of algorithms. Figure 10 shows the class structure of the pattern corresponding to a non-reflective implementation.

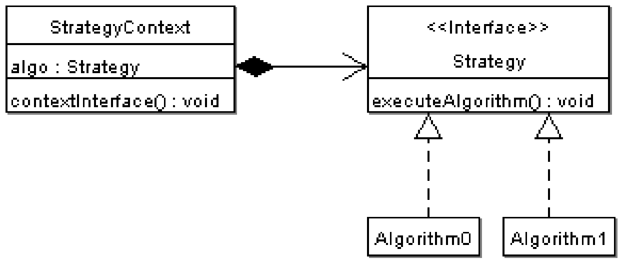

**Figure 10. Class structure of the Strategy pattern**

Figure 11 shows the class structure of the Strategy pattern corresponding to an alternative implementation using proxies. The context class is replaced by two classes: `ProxyFactory` and `StrategyHandler`; the latter implements `InvocationHandler` of the `java.lang.reflect` package and corresponds to the meta-object. There is no direct association between application strategy classes at the base level and classes representing context objects at the meta-level. The coupling of strategies to context objects is established at runtime with the help of a `ProxyFactory`. We note that by changing the mechanisms of associating strategies to their contexts, the requirements of the Strategy pattern are preserved, i.e., encapsulating the family of algorithms and making it changeable independent from the context. In addition, contexts are made generic, reusable and independent of the algorithms.

With the reflection model supported by the proxies, separating algorithms from contexts is implemented as a direct consequence of the separation between base- and meta-level. Method interception of meta-objects suffices and no manual changes to the static class structure are needed. There is otherwise no coupling at the class level between algorithms and their contexts. This is not the case in the *static* implementation of Figure 10; a context object *contains* a strategy so that method invocations to execute the various algorithms can be delegated.

The implementation using proxy objects is depicted in Figure 12. The class `StrategyHandler` makes use of the dispatching mechanism provided by dynamic proxies to invoke the `executeAlgorithm()` methods on base objects of type `Strategy`.

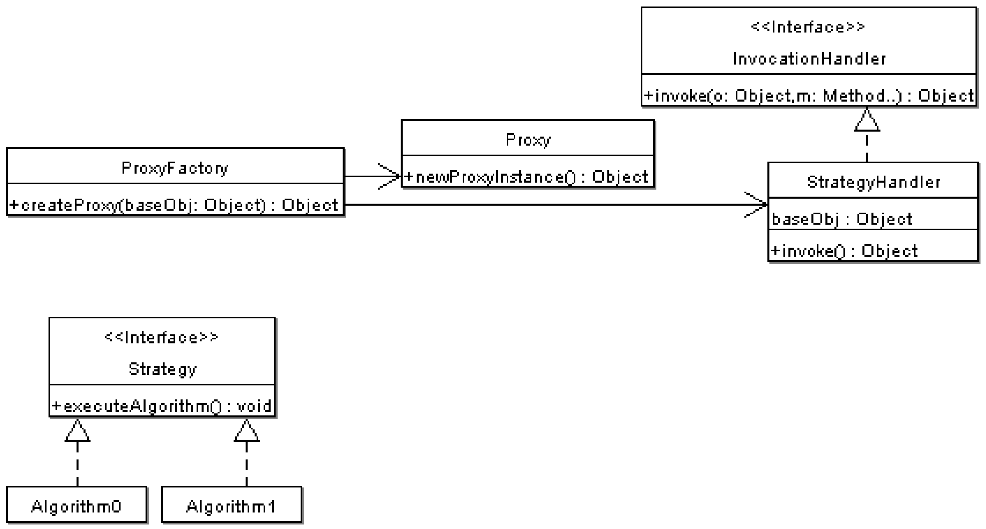

**Figure 11. Class structure of the Strategy pattern separating base level from meta-level**

The class diagrams in Figures 10 and 11 show that proxy implementation of the pattern exhibits less coupling when compared with direct implementation. In the latter case, a context is replaced by a meta-object to which control is dispatched by invoking strategy methods on corresponding proxy objects. A context is not only explicitly separated from the algorithms, but is also independent of them; it is generic and can be reused irrespective of how the algorithms and their methods are implemented.

```
import java.lang.reflect.*;

public class ProxyFactory {
  public Object createProxy(Object algo) throws Throwable {
    // as in Figure 3. The invocation handler is a StrategyHandler
  }
}
class StrategyHandler implements InvocationHandler {
  private Object base_obj;
  public StrategyHandler (Object base_obj) { this. base_obj=base_obj; }
  public Object invoke(Object p, Method m, Object[] args) throws Throwable {
    Object result = null;
    try {
      result = m.invoke(base_obj, args);
    } catch (InvocationTargetException e) { throw e.getTargetException();}
    return result;
  }
}
```

**Figure 12. The meta-level part of the Strategy pattern implementation**

## The State Pattern

The State pattern allows an object to alter its behaviour when its internal state changes. The State pattern has similarities with the Strategy pattern; both are examples of composition and delegation. The difference lies in the intent; a state object encapsulates a state-dependent behaviour and possibly a state transition rule while a strategy object encapsulates an algorithm.

Separating states from their contexts follows the same procedure as that applied to the Strategy pattern. State objects reside in the base level and contexts are meta-objects controlling state transitions. A State transition scheme is defined by the application at the base level. The context object at the meta-level identifies current states including the initial state and invokes the state method on these objects.

Figures 13 and 14 show an implementation of the state pattern using the reflection model based on proxies. The meta-object (`StateHandler`) controls state transitions by holding the current state and setting it according to the value of the received base object (lines 19-32).

```
import java.lang.reflect.*;                                                    //1
                                                                               //2
public class ProxyFactory {                                                    //3
  public Object createProxy(Object base_obj) throws Throwable {                //..
    // as in Figure 3. The invocation handler is a StateHandler
  }
}                                                                              //12
class StateHandler implements InvocationHandler {                              //13
  private Object base_obj;                                                     //14
  private Object curr_state=null;
  public StateHandler(Object base_obj) {
    this. base_obj=base_obj;
  }
  public Object invoke(Object proxy, Method m, Object[] args) throws Throwable { //19
    Object result = null;                                                      //20
    try {                                                                      //21
      Class sc=base_obj.getClass().getSuperclass();                            //22
      if (curr_state==null) {                                                  //23
        result=m.invoke(base_obj, args);                                       //24
        curr_state=sc.getField("curr_state").get(base_obj);                    //25
      } else  {
        result = m.invoke(curr_state, args);
        curr_state=sc.getField("curr_state").get(curr_state);
      }
    } catch (InvocationTargetException e) { throw e.getTargetException();}     //30
    return result;                                                             //31
  }                                                                            //32
}                                                                              //33
```

**Figure 13. The handler implementation of the State pattern**

```
interface Istate { public void foo(); }

abstract class State {
  public IState curr_state;
  public void change_state(IState is) { curr_state=is; }
}
class State_0 extends State implements IState {
  public void foo() {
    System.out.println("current state: State_0 ");
    change_state(new State_1());
  }
}
class State_1 extends State implements IState {
  public void foo() {
    System.out.println("current state: State_1 ");
    change_state (new State_2());
  }
}
class State_2 extends State implements IState {
  public void foo() {
    System.out.println("current state: State_2");
    change_state(new State_0());
  }
}
public class StateDemo {
  public static void main(String [] args) throws Throwable {
    ProxyFactory pf=new ProxyFactory();
    IState is=(IState)pf.createProxy(new State_2());
    is.foo(); is.foo(); is.foo(); is.foo(); is.foo(); is.foo();
  }
}
```

**Figure 14. A client dispatching requests to the State pattern handler**

In a traditional implementation of the State pattern, the meta-program is replaced by a context class(es). The context class maintains a state object (of type `IState`) and delegates state-specific requests to this object. Unlike the proxy implementation where the context is decoupled from state objects and coupling becomes active only when requested at runtime, the coupling between contexts and state objects in a non-reflective implementation holds during the entire lifetime of the context object.

## The Iterator Pattern

The Iterator pattern provides a way of accessing the elements of an aggregate object sequentially without exposing its underlying representation. Java provides a number of iterator classes in the java.util package as part of the language collections framework. These classes support polymorphic iteration by providing traverse and access operations independent of the aggregate internal structure. Java Iterators separate the generic control structure of the pattern from specifics of their clients. Consequently, the objectives of applying the reflection model to this pattern are met by the Java library class implementations.

## The Visitor Pattern

The Visitor pattern separates object structures from operations acting on the objects and thus allows the addition of operations to classes without changing those classes. To achieve this, Visitor uses a double-dispatch technique where dispatching control to a method depends on the name of the request and the types of two receivers, i.e., the object and its visitor.

The Visitor pattern could be considered as a substitute for the double-dispatch mechanism unsupported by mainstream object-oriented languages such as C++ and Java. As observed by Gamma et al. [8], languages that support multiple dispatching (for example CLOS) lessen the need for this pattern.

The class diagram of the Visitor pattern is depicted in Figure 15 and Figure 16 shows the corresponding implementation. We note the dependency between objects and their visitor; the objects' `accept()` method takes a `IVisitor` type as parameter and a visitor operation (`visit()` method) has a parameter of type `ITreeNode`. There are two different `ITreeNode` class types, each of which implements the `accept()` operation, thus allowing the visitor to act upon its objects. The `Visitor` class provides implementations of visiting operations for each `ITreeNode` object type. In running the pattern, the nodes can be traversed in which each object is operated upon by invoking the `accept()` method.

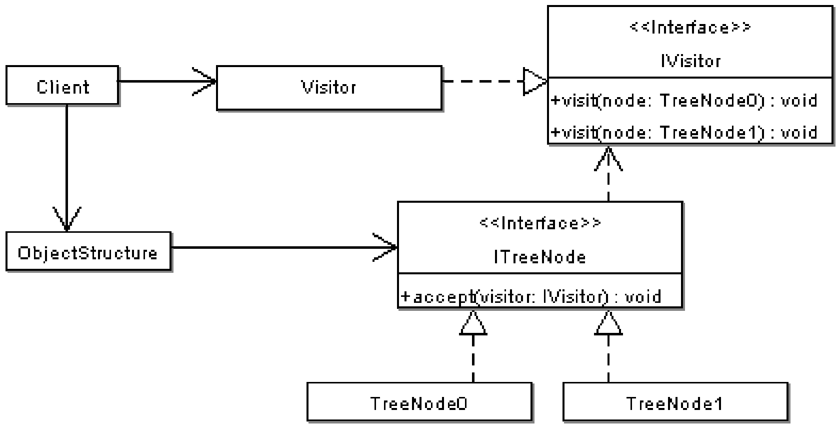
Client
Visitor
<<Interface>>
IVisitor
+visit(node: TreeNode0) : void
+visit(node: TreeNode1) : void
ObjectStructure
<<Interface>>
ITreeNode
+accept(visitor: IVisitor) : void
TreeNode0
TreeNode1

**Figure 15. Class structure of the Visitor pattern**

We can implement the double-dispatch technique by using the dispatch mechanism provided by dynamic proxies. Node objects are separated from the visitor by assigning them to the base level; proxy objects residing on the meta-level take the role of visitors. Once an `accept()` method is invoked on an `ITreeNode` object, control is dispatched to the meta-level where reflective code implemented by the meta-object is executed. Figure 17 shows the class structure of the Visitor pattern according to this view. Note that the classes of proxy objects are completely decoupled from application classes at the base level.

```
interface ITreeNode { public void accept(IVisitor v); }

class TreeNode_0 implements ITreeNode {
  public void accept(IVisitor v) { v.visit(this); }
  public String visit_treeNode_0() { return "visiting TreeNode_0"; }
}
class TreeNode_1 implements ITreeNode {
  public void accept(IVisitor v) { v.visit(this); }
  public String visit_treeNode_1() { return "visiting TreeNode_1: "; }
}

interface IVisitor {
  public void visit(TreeNode_0 tn_0);
  public void visit(TreeNode_1 tn_1);
}
class Visitor implements IVisitor {
  public void visit(TreeNode_0 n_0){
    System.out.println(n_0.visit_treeNode_0());
  }
  public void visit(TreeNode_1 n_1){
    System.out.println(n_1.visit_treeNode_1());
  }
}

public class VisitorPattern {
  static ITreeNode[] tree=
        { new TreeNode_0(), new TreeNode_1() };
  public static void main( String[] args ) {
    IVisitor aVisitor=new Visitor();
    for (int i=0; i < tree.length; i++) tree[i].accept(aVisitor);
  }
}
```

**Figure 16. An implementation of the Visitor pattern**

Figure 17 suggests that, at the base level, the method `accept()` is made independent of the visitor. As a result, `ITreeNode` classes provide a default (an empty) implementation of the method. The `accept()` method is invoked on proxy objects triggering the visitor's operation implemented at the meta-level. The result is the same as in the static case. The differences lie in the implementation of the visitor object as well as in its activation.

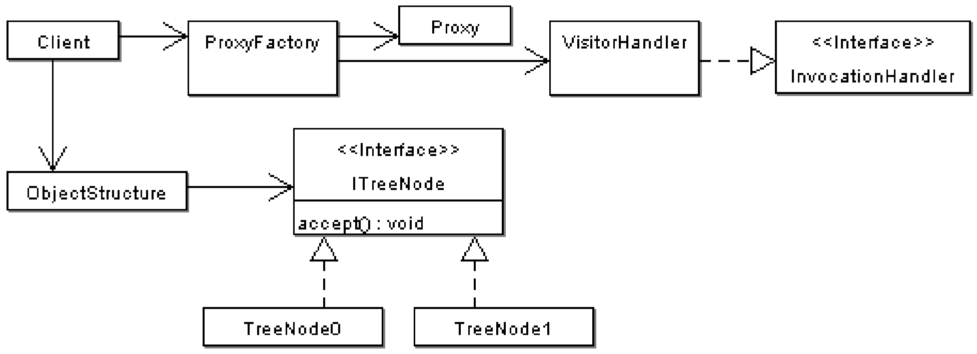

**Figure 17. Class structure of the Visitor pattern separating base and meta-levels**

Figure 18 shows an extract of the corresponding Java implementation of the pattern. Instances of class `VisitorHandler` represent visitor objects (lines 14-33) operating on base objects by intercepting `accept()` method invocations on them. Reflection is used to invoke the visit methods on base objects. Note that in executing the meta-code, meta-objects assume that objects at base level provide visiting methods whose names start with “visit”.

```
import java.lang.reflect.*;                                                      //1
                                                                                 //2
public class ProxyFactory {                                                      //3
  public Object createProxy(Object base_object) throws Throwable {               //4
   // ... The invocation handler is a VisitorHandler                             //..
  }                                                                              //11
}                                                                                //12
                                                                                 //13
class VisitorHandler implements InvocationHandler {                              //14
  private Object tnode;                                                          //15
  public VisitorProxy (Object tnode) {                                           //17
    this. tnode = tnode;                                                         //18
  }                                                                              //19
  public Object invoke(Object p, Method m, Object[] args) throws Throwable{      //20
    Object result = null;                                                        //21
    try {                                                                        //23
      Method[] meths=tnode.getClass().getDeclaredMethods();                      //24
      for (int i=0; i < meths.length; i++) {                                     //25
        if (meths[i].getName().startsWith(“visit”)) {                            //26
          result=meths[i].invoke(tnode, null);                                   //27
        }                                                                        //28
      }                                                                          //29
    } catch (InvocationTargetException e) { throw e.getTargetException(); }     //30
    return result;                                                               //31
  }                                                                              //32
}                                                                                //33
```

**Figure 18. An implementation of the Visitor pattern at the meta-level**

## The Observer Pattern

Observer defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. This is equivalent to the Model-View-Controller (MVC) pattern, often used in constructing graphical user interface or GUIs.

Java provides a number of abstractions that allow direct implementation of the pattern. There is, on the one hand, the class tuple (`Observable`, `Observer`) in the `java.util` package. Observable objects keep track of interested observers. Once a change of state of an observable object takes place, all observers are notified and updated automatically. On the other hand, there are the JavaBeans classes `PropertyChangeEvent`, `PropertyChangeListener` and `PropertyChangeSupport`, which support active JavaBeans properties. Active properties are characterized by the firing of an event when the state of the object changes; for example, by invoking set methods.

In Figure 19, we implement the Observer pattern using JavaBeans classes. By adapting this implementation to our reflection model, we provide a generic coding of the event-trigger mechanism independent of the application object (the observable). The class `BeanHandler` is passed a base object as a parameter together with a listener of type `PropertyChangeListener`. Using Java’s introspection API, the handler derives the property name and the corresponding get method. The old value is obtained by invoking the get method (line 21) on the base object and the new value is obtained after the set method is invoked (lines 27 and 28). With both values and the property name at hand, an event of change of state can be fired. The listener which is added upon the creation of the meta-object (line 10) can be informed using the delegate class `support` of type (`PropertyChangeSupport`-line 39). `IPropertyChangeSupport` provides a generic interface for adding/removing of listeners and firing events (lines 40-48)

```
import java.lang.reflect.*;                                                     //1
                                                                                //2
public class BeanHandler implements InvocationHandler, IPropertyChangeSupport {  //3
  private Object base_obj;
  private Object listener;

  public BeanProxy(Object base_obj, Object listener) {
    this.base_obj=base_obj;
    this.listener=listener;
    (this).addPropertyChangeListener((PropertyChangeListener)listener);          //10
  }
  public Object invoke(Object proxy, Method m, Object[] args) throws Throwable { //12
   Object result = null;
    if (m.getName().startsWith("set")) {
      String property=m.getName().substring("set".length());
      Object oldVal=null; Object newVal=null;
      Method[] meths=base_obj.getClass().getDeclaredMethods();
      Method getMethod=null;
      for (int i=0; i<meths.length; i++) {
        if ((meths[i].getName().startsWith("get")!=-1)
                             &&(meths[i].getName().indexOf(property)!=-1)) {
          try { oldVal=(Integer)meths[i].invoke(base_obj, null);                 //21
          } catch (InvocationTargetException e) { throw e.getTargetException(); }
          getMethod=meths[i];
        }
      }
      try {
        result = m.invoke(base_obj, args);                                       //27
        newVal=getMethod.invoke(base_obj, null);                                 //28
      } catch (InvocationTargetException e) { throw e.getTargetException();}     //29
      (this).firePropertyChange(property, oldVal, newVal);                       //30
    }
    else {   // not a set method, proceed normally
      try { result = m.invoke(base_obj, args);
      } catch (InvocationTargetException e) { throw e.getTargetException();}
    }
    return result;
  }

  private PropertyChangeSupport support = new PropertyChangeSupport(this);       //39
  public void addPropertyChangeListener(PropertyChangeListener ls){              //40
    support.addPropertyChangeListener(ls);
  }
  public void removePropertyChangeListener(PropertyChangeListener ls) {
    support.removePropertyChangeListener(ls);
  }
  public void firePropertyChange(String ppy, Object oldval, Object newval) {
    support.firePropertyChange(ppy, oldval, newval);
  }                                                                              //48
}                                                                                //49
```

**Figure 19. An implementation of Observer pattern using JavaBeans classes**

The generic Observer pattern implementation of (`BeanHandler`) shows the power of our reflection model in supporting code reuse and less coupling. More details on the Observer implementation using JavaBeans mechanisms can be found in [11].

## The Command Pattern

The Command pattern encapsulates a request as an object. Command decouples client objects from objects having the knowledge to perform the requested operations.

Adapting the Command pattern to the proxies' reflective model is straightforward, due to the fact that the model supports decoupling of objects and classes. Clients can be identified with base objects and commands as meta-objects residing at the meta-level. The meta-objects are assisted by other objects having the knowledge to perform the required operations. Clients' requests are dispatched to invocation handler (meta-objects), which pass on the request to appropriate objects for manipulation.

Figure 20 shows an example of how two commands are implemented (`LightOnCommand` and `LightOnCommand`) together with a `CommandHandler` controlling the execution of these commands. Switching lights on or off depends on clients' input dispatched to the handler through a proxy object. Figure 21 is an example of a client using a proxy to pass its requests to a command controller residing at the meta-level. The coupling between client objects and commands takes places at runtime and is temporal; once the commands are executed the coupling ceases to exist.

```
import java.lang.reflect.* ;

public class ProxyFactory {
  public Object createProxy(Object base_obj) throws Throwable {
    // .... . The invocation handler is a CommandHandler
  }
}

class CommandHandler implements InvocationHandler {
  private Object swit;
  private Light light;
  public CommandHandler(Object swit) {
    this.swit=swit;
    light=new Light();
  }
  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    Object result = null;
    try {
      if (method.getName().equals("switchOnCommand")) {
         new LightOnCommand(light).execute();
      }
      else if (method.getName().equals("switchOffCommand"))  {
         new LightOffCommand(light).execute();
      }
      else result=method.invoke(swit, args);
    } catch (InvocationTargetException e) { throw e.getTargetException(); }
    return result;
  }
}

interface ICommand { public abstract void execute(); }
class Light {
  public void turnOn() { System.out.println("Light is on "); }
  public void turnOff() { System.out.println("Light is off"); }
}
class LightOnCommand implements ICommand {
  private Light myLight;
  public LightOnCommand (Light L) { myLight  =  L; }
  public void execute() { myLight . turnOn( ); }
}
class LightOffCommand implements ICommand {
  private Light myLight;
  public LightOffCommand (Light L) { myLight  =  L; }
  public void execute() { myLight . turnOff( ); }
}
```

**Figure 20. CommandHandler controls execution of commands**

```
interface IRequest {
  public void switchOnCommand ();
  public void switchOffCommand ();
}
class Switch implements IRequest {
  public void switchOnCommand() { }
  public void switchOffCommand() { }
}
public class CommandDemo {
  public static void main(String[] args) throws Throwable {
    ProxyFactory pf=new ProxyFactory();
    IRequest ireq=(IRequest)pf.createProxy(new Switch());
    ireq.switchOnCommand(); ireq.switchOffCommand(); ireq.switchOffCommand();
  }
}
```

**Figure 21. Commands are dispatched to the meta-level using a proxy object**

The advantages of implementing the Command pattern using dynamic proxies are less coupling at the class (and object) level and promotion of modularity resulting from the separation of clients from commands. The command code is autonomous, self-contained and independent of clients' specifics.

## The Chain of Responsibility Pattern

The Chain of Responsibility pattern avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Receiving objects are chained and the request is passed along the chain until an object handles it.

As for Command, Chain of Responsibility promotes looser coupling between classes. It forwards requests along a chain of classes whereas the Command pattern forwards a request to a specific class. Adapting the pattern to the proxies' reflective model follows the same as the procedure applied to the Command pattern. Request senders are interpreted as base objects and the chain of handler resides at the meta-level. The proxy's invocation handler receives clients' requests and implements the chaining operation with the support of receiver objects.

## The Mediator Pattern

The Mediator pattern defines an object that encapsulates how a set of objects interacts. The pattern promotes loose coupling by keeping objects from referring to each other explicitly.

We can minimise coupling further by applying our reflective model to Mediator. The role of the mediator object can be implemented at the meta-level. The meta-object keeps a list of all created objects and implements the interaction strategy by providing necessary operations needed to ensure that the objects work properly together. We note that the objects are instantiated at the application level and passed to the *same* meta-object upon creation of corresponding proxies.

Applying the reflection model promotes loose coupling of Mediator further; the set of objects whose interactions are to be controlled by the Mediator but are passed at runtime and have no reference to the Mediator (of type `InvocationHandler`) at the class level. In a non-reflective implementation of the pattern, there is an additional coupling resulting from the dependency of interacting objects on the Mediator at compile-time.

## The Interpreter Pattern

The Interpreter pattern defines a representation for the grammar of a language along with an interpreter that uses the representation to interpret sentences in the language. Implementing the Interpreter pattern requires knowledge of compiler techniques including parsing the language expressions, interpreting the resulting syntax tree into actions and then executing the actions to be exercised. Error checking will also be required to handle grammatical statements with incorrect syntax.

The Interpreter pattern is useful in cases where a *simple* compiler is needed to handle a particular set of expressions of an arbitrary application. The compiler construction process consists of several consecutive steps (scanning, parsing, code generation etc) normally executed in a certain order. Inserting proxies in the compilation process as a result of applying our reflection model is inadequate, because it complicates the interpretation process with no apparent benefits.

**The Adapter Pattern**

The Adapter pattern converts the interface of one class into that of another. The intention is to convert the interface of a class to match the requirements of a client class.

There are two ways to implement Adapter: through inheritance and through composition. The two approaches are termed class adapter and object adapter, respectively. The non-conforming class is referred to as the adaptee and the new class, which converts its interface, as the adapter. In the case of inheritance, the adapter is defined as a sub-class of the adaptee and the methods required to match the desired interface are added to it. In the case of composition, the adaptee is included inside the adapter and acts as a delegate to method calls invoked on adapter objects.

Adapters are intermediary entities between adaptees and client objects. Proxy objects can play this intermediary role of adapters. The difference to the class and object adapter implementations is that a dynamic proxy (as adapter) acquires the target interface at runtime. It is independent of this interface and uses reflection to identify method calls. The proxy depends on the adaptee to implement the required functionality, as in the class and object adapter implementation. Furthermore, the proxy implementation does not substitute the original target class with the adapter class, but keeps this class and represents its objects at runtime.

Figure 22 shows an example of how our model can be used to implement the Adapter pattern in such a way that a proxy object plays the role of an adapter. In this implementation, the handler uses a `java.util.Vector` object (the adaptee) as delegate to implement the required functionality. Figure 23 shows an example of how a target class interface (`BaseStack`) can be converted into a vector interface by passing the client request to the meta-object using a proxy. We note that the only change required to adapt the default implementation represented by `BaseStack` is to divert the method invocation to the proxy object.

```
import java.lang.reflect.*;

public class ProxyFactory {
  public Object createProxy(Object base_obj) throws Throwable {
    // as in Figure 3. The invocation handler is a StackHandler
  }
}

class StackHandler implements InvocationHandler {
  private Object stack;
  private java.util.Vector adaptee;
  public StackHandler(Object stack) {
    this.stack=stack;
    adaptee=new java.util.Vector();
  }
  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    Object result = null;
    try {
        if (method.getName().equals("push")) adaptee.addElement(args[0]);
        else if (method.getName().equals("pop")) {
          Object le = adaptee.lastElement();
          adaptee.removeElementAt(adaptee.size() - 1);
          result=le;
        }
        else if (method.getName().equals("top")) result=adaptee.lastElement();
        else result=method.invoke(stack, args);
    } catch (InvocationTargetException e) { throw e.getTargetException();}
    return result;
  }
}
```

**Figure 22. A Vector object as adaptee**

```
import java.util.*;
interface IStack {
  public void push(Object v);
  public Object pop( );
  public Object top( );
  public boolean empty( );
}
class BaseStack implements IStack {
public void push(Object v) { }
  public Object pop( ) { return null; }
  public Object top( ) { return null; }
  public boolean empty( ) { return true; }
}

public class AdapterDemo {
  public static void main(String [] args) throws Throwable {
    ProxyFactory fac=new ProxyFactory();
    IStack stack=(IStack)fac.createProxy(new BaseStack());
    stack.push( new Integer(100)); stack.push( new Integer(200));
    System.out.println(""+stack.top());    System.out.println(""+stack.top());
  }
}
```

**Figure 23. Adapting default implementation of a stack using a proxy**

**The Decorator Pattern**

The Decorator pattern attaches additional responsibilities to an object dynamically. It provides a flexible alternative to sub-classing for extending functionality. Decorator and Adapter are wrapper patterns. While decorator wraps and adds new responsibilities to a core object without changing its interface, adapter changes the adaptee's interface.

Decorator can be easily adapted to the proposed reflection model. Core objects residing at the base level can be *decorated* by attaching them to meta-objects. Meta-objects extend the functionalities of their referents before delegating control back to them. The examples in Figures 22 and 23 and used to explain the role of proxy objects in implementing the Adapter pattern, can also serve as a good example for the Decorator pattern.

**The Memento Pattern**

The Memento pattern stores the current state of an object in a memento object. The object's state can be later restored without exposing the internal representation of the object to the outside world.

We can adapt the Memento pattern to the proposed reflection model by letting meta-objects implement the undo and save operations invoked on their referents. Here, we assume that the internal states of base objects need to be stored by mementos. When the undo operation is invoked, a meta-object asks the base object to retrieve stored data from its memento. When the save operation is invoked, a meta-object allows its referent to create a new memento object and store data in the newly created object. The class relationships between base objects and their mementos remain the same as in the non-reflective case.

The advantage of implementing Memento using dynamic proxies is the separation of application objects from objects manipulating the states of these objects, thus increasing the modularity level. This separation is particularly useful when save and undo are part of typical database transactions. In this case, meta-objects can act as buffers holding application objects states and as initiators of database actions, thus separating object state manipulations from object data.

Figure 24 shows an example of implementation of the undo and save operations using our proxy-based reflection model. The meta-object (of type `MementoHandler`) holds references to the base object (a `Point`) and its memento (a `PointMemento`) passed as parameters upon creation (16-19). When `save()` or an `undo()` method is invoked at the base level, control is dispatched to the meta-object. In case of `save()`, a memento object is created and the base object x and y-fields are set (lines 22-26). In case of `undo()`, the fields are retrieved by invoking `setMemento((PointMemento)mem)` and the base object is returned with the retrieved values (lines 27-31).

Figure 25 shows the application classes including `Point` and `PointMemento`. `MementoDemo` represents a simple example showing how to invoke the save and undo operations through a proxy object.

```
import java.lang.reflect.*;                                                    //1
                                                                               //2
public class ProxyFactory {                                                    //3
  public Object createProxy(Object base_obj, Object mem) throws Throwable {    //4
    // ...                                                                     //..
  }
}                                                                              //12
class MementoHandler implements InvocationHandler {
  private Object base_obj;
  private Object mem;
  public MementoHandler(Object base_obj, Object mem) {                         //16
    this.base_obj=base_obj;
    this.mem=mem;
  }                                                                            //19
  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    try {
      if (method.getName().equals("save")) {                                   //22
        mem=(PointMemento)(((Point)base_obj).createMemento());
        ((Point)base_obj).setx(new Integer(((Point)base_obj).getx()).intValue());
        ((Point)base_obj).sety(new Integer(((Point)base_obj).gety()).intValue());
      }                                                                        //26
      else if (method.getName().equals("undo")) {                              //27
        if (mem==null) throw new Exception("No Mememnto object ...");
        ((Point)base_obj).setMemento((PointMemento)mem);
        return ((Point)base_obj);
      }                                                                        //31
    } catch (InvocationTargetException e) { throw e.getTargetException();}
    return null;
  }
}
```

**Figure 24. Undo and save operation implemented at the meta-level**

```
interface IPoint {
    public void setx(int x);
    public void sety(int y);
    public int getx();
    public int gety();
}
interface IMemento { public void save(); public void undo(); }
class Point implements IPoint, IMemento {
   private int x = 0; private int y = 0;
   public Point() { this.x = 0; this.y = 0; }
   public Point(int x, int y) { this.x = x; this.y = y; }
   public void setx(int x) { this.x = x; }
   public void sety(int y) { this.y = y; }
   public int getx() { return x; }
   public int gety() { return y; }
   public String toString() { return ("("+x+", "+y+")"); } ;
   public PointMemento createMemento() { return new PointMemento(x, y); }
   public void setMemento (PointMemento pm) { x=pm.x; y=pm.y; }
   public void save () { System.out.println("Saving the point ..."); }
   public void undo () { System.out.println("Undoing the point ..."); }
}
class PointMemento {
   private int x = 0;    private int y = 0;
   PointMemento(int x, int y) { this.x=x; this.y=y; }
}

public class MementoDemo {
  public static void main(String [] args) throws Throwable {
    Point pt=new Point();
```

```
      ProxyFactory pf=new ProxyFactory();
      IMemento im=(IMemento)pf.createProxy(pt, new PointMemento(0, 0));
      pt.setx(100);  pt.sety(50);  im.save();
      System.out.println(pt.toString());
      pt.setx(-2); pt.sety(-6); im.undo();
      System.out.println(pt.toString());
    }
  }
```

**Figure 25. Application classes of the Memento pattern**

## The Bridge Pattern

The Bridge pattern is meant to decouple an abstraction from its implementation so that the two can vary independently. Abstraction represents the interface through which clients interact with the system.

The Bridge pattern supports the principle of abstraction in software engineering, according to which implementation details are separated from and hidden behind application's interface. The proposed reflection model supports the principle of abstraction. However, the model allows clients to modify application behaviour through a meta-interface. The modification is carried out without modifying the default implementation.

Adapting the Bridge pattern to the proposed reflection model is useful only if the new feature of the model, namely, open implementation, can be used effectively and the benefits gained outweigh the complexity of adding this new feature.

## The Composite Pattern

The Composite pattern composes objects into tree structures to represent whole-part hierarchies and allows individual objects (leaves) and composites (nodes) to be treated uniformly. The pattern is used to describe a recursive composition in such a way that simple and compound objects are treated equally.

Figure 26 shows the class structure of the Composite pattern. The programming units `IComponent` and `Component` represent abstractions providing the basic functionality of a recursive structure and allow a uniform treatment of simple and compound objects. The `Leaf` class represents simple objects and hence inherits the default behaviour defined by its super-class, `Component`. By default, a `Component` cannot add to, remove from, or navigate through the object structure. Class `Leaf` defines individual object attributes and how objects specify themselves. Class `Composite` represents compound objects and defines the functionality needed to manipulate and navigate through the objects' structure in the form of a stack. The stack is constructed recursively through the `add()` method. The recursive structure of `Composite` is represented by a self-association as shown in Figure 26. Specifying a composite object requires traversing its structure and, at each entry, calling the corresponding `specify()` method. The `remove()` method removes the first object from the list. Figure 27 shows an extract of a Java implementation of the pattern without using meta-level facilities provided by the proxies.

To adapt the static structure of the Composite pattern to the reflective two-level architecture, we need to identify the generic aspects of the pattern and separate them from those that are application specific. Simple objects with their attributes and specifications are application dependent. Recursive structures are abstract data types and are inherently generic. Following this view, simple and compound objects can be treated uniformly and classified as base and meta-objects, respectively. Figure 28 shows the class structure of the Composite pattern according to this classification. Note that `CompositeHandler` refers to itself reflecting the recursive structure of the stack. As in previous cases, the class diagram shows no coupling between classes of the two levels.

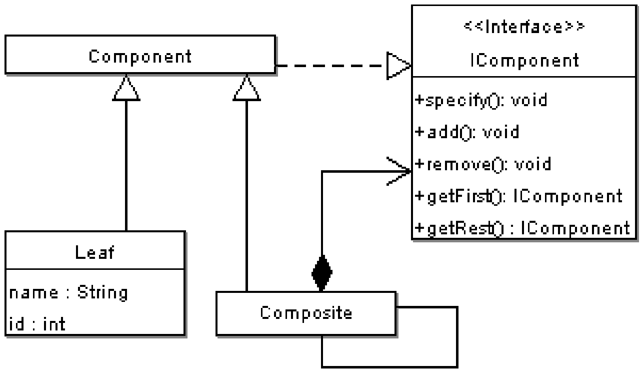

**Figure 26. Class structure of the Composite pattern**

```
interface IComponent {                                                          //1
  public void specify();                                                        //2
  public void add(IComponent c) throws Exception;                               //3
  public void remove() throws Exception;                                        //4
  public IComponent getFirst();                                                 //5
  public IComponent getRest();                                                  //6
}                                                                               //7
class Component implements IComponent {                                         //8
 // provides a default implementation of IComponent. Throws Exceptions          //9
 // for add and remove and returns nulls for getFirst and getLast               //10
}                                                                               //11
class Leaf extends Component {                                                  //12
  public String name;                                                           //13
  public int id;                                                                //14
  public Leaf(String name, int id) { this.name=name; this.id=id; }              //15
  public void specify() { // specify a leaf ... }                               //16
}                                                                               //17
class Composite extends Component {                                             //18
  // implements stack structure recursively                                     //19
  IComponent first;
  IComponent rest;
  Composite cons; // will be first defined by calling add
  public Composite() { first=null; rest=null; }
  public Composite(IComponent f, IComponent r) {  this.first = f; this.rest = r; }
  public void specify() {
    IComponent en=cons.getFirst();
    IComponent re=cons.getRest();
    while (en!=null && re!=null) {
      en.specify();
      en=(re.getFirst());
      re =re.getRest();
    }
  }
  public void add(IComponent o) {
   if (cons==null) cons=new Composite(); cons=new Composite(o, cons);
  }
  public void remove () { // for simplicity removes 1st object
    cons.first=(cons.getRest()).getFirst();
    cons.rest =(cons.getRest()).getRest();
  }
  public IComponent getFirst() { return this.first; }
  public IComponent getRest() { return this.rest; }
}                                                                               //43
public class DemoComposite {                                                    //44
  public static void main(String [] args) {                                     //45
    // create a Composite object, add Leaf objects & manipulate                 //46
    Composite con=new Composite();
    con.add(new Leaf("xxxxx", 01));
    con.add(new Leaf("yyyyy", 02));
    con.specify(); con.remove(); con.specify();
  }
}                                                                               //52
```

**Figure 27. An implementation of the Composite pattern**

The base level implementation of the Composite pattern can be deduced from the UML diagram of Figure 28 and described as follows. The abstraction `IComponent` which defines the recursive structure and represents simple and compound components, remains the same as in the static case. Similarly, class `Component` defining default behaviour of components and class `Leaf`, which, implements the behaviour of a simple component object also remain the same as before. To run the pattern application, a meta-object is reified upon instantiating a proxy using the factory method of `ProxyFactory`. The result is identical to that of the static case. The difference is that, in the dynamic case, manipulating a compound object is realized at the meta-level in a generic manner without reference to the base class types, `IComponent`, `Component` and `Leaf`.

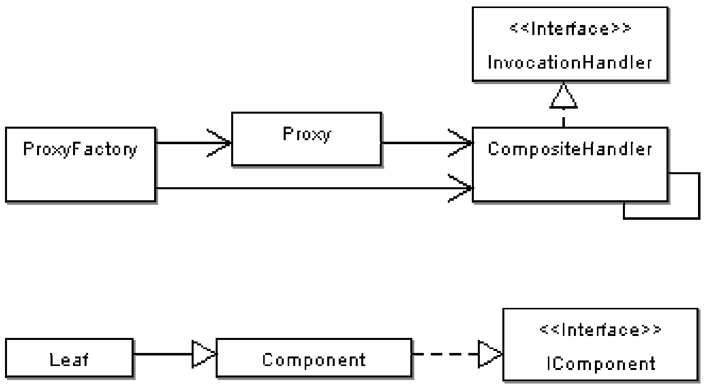

**Figure 28. Class structure of the Composite pattern separating base and meta-objects**

Figure 29 shows an extract of the Composite pattern implementation at the meta-level. The class `CompositeHandler` represents compound objects (lines 14-35). As the `Composite` class in the static case, it implements all the methods of the component interface. The major difference is that `CompositeHandler` makes no reference to base object types. To manipulate the recursive structure of a compound object, the methods of adding or removing simple objects, navigating through the structure and specifying the entries are intercepted and delegated to the generic `invoke()` method (lines 22-34). With the `specify()` method, the recursive structure is traversed and at each entry the corresponding method of simple objects is called back.

```
import java.lang.reflect.*;                                                      //1
                                                                                 //2
public class ProxyFactory {                                                      //3
  public Object createProxy(Object base_obj) throws Throwable {                  //..
   // as in Figure 3. The invocation handler is a CompositeHandler
  }
}                                                                                //12
                                                                                 //13
class CompositeHandler implements InvocationHandler {                            //14
  private Object base_obj;                                                       //15
  CompositeProxy cons;                                                           //16
  // define first and rest variable ...                                          //17
  public ContextComposite (Object base_obj) {                                    //18
    this. base_obj=base_obj;                                                     //19
    cons=new ContextComposite ();                                                //20
  }                                                                              //21
  public Object invoke(Object p, Method m, Object[] args) throws Throwable {     //22
    Object result = null;                                                        //23
    try {                                                                        //24
      if (m.getName().equals(add)) {                                             //25
        cons=new CompositeProxy(args[0], cons);                                  //26
      }                                                                          //27
      else if (m.getName().equals(specify)){ // as in static case }              //28
      else if (m.getName().equals(remove)) { // as in static case }              //29
      else if (m.getName().equals(getFirst)) cons.getFirst();                    //30
      else if (m.getName().equals(getRest))  cons.getRest();                     //31
      } catch (InvocationTargetException e) { throw e.getTargetException(); }
      return result;
    }
  }
```

**Figure 29. The meta-level part of the Composite pattern implementation**

The advantages of separating the generic part of the Composite pattern from the application specific part are less coupling, reuse of the meta-code, and support for separation of concerns (base-level/meta-level). The combination of these features allows for a better understanding of design and for potentially reduced maintenance costs.

### The Facade Pattern

The Facade pattern provides a unified interface to a set of interfaces in a sub-system. Facade defines a higher-level interface that makes the subsystem easier to use. This can be used to simplify a number of complicated object interactions into a single interface.

Facade can be seen as an extended form of the Adapter pattern; while an adapter wraps one object, facade wraps a number of existing objects. As in the case of Adapter, a dynamic proxy can play the role of an intermediary between client and a *set* of objects. In this case, objects are attached to the facade at runtime. As in the case of Adapter, the advantages of using a dynamic proxy as facade are flexibility and less coupling.

## The Flyweight Pattern

The Flyweight pattern uses sharing to support large numbers of fine-grained objects efficiently. The participants involved in the pattern are the `Flyweight`, the `ConcreteFlyweight`, the `UnsharedConcreteFlyweight`, the `FlyweightFactory` and the `Client`. `ConcreteFlyweight` implements the `Flyweight` interface and stores intrinsic states independent of its objects context. A `ConcreteFlyweight` object must be shareable whereas an `UnsharedConcreteFlyweight` not. The `FlyweightFactory` serves to deliver particular flyweights requested. The `Factory` is passed certain properties and returns the requested flyweight if it already exists; otherwise it creates and returns a new flyweight. In addition, there is the `Context` class, which acts as a repository of extrinsic state. When creating a new object, a `Client` assigns a flyweight to the object and computes its extrinsic state.

We note that the role of `Context` together with `FlyweightFactory` is to control the flow of flyweights required by a `Client`. On the other hand, `Flyweight`, `ConcreteFlyweight` and `UnsharedConcreteFlyweight` provide the data structure whose storage is to be efficiently manipulated.

We can employ the reflection model and separate the control classes from the structural classes. However, because of the large number of objects involved, we expect computation time to increase due to the reflection overhead caused by the proxy mechanism. In addition, employing the reflection model does not reduce the level of coupling, because the factory must know the flyweight structure (i.e., type) in order to decide whether to create a new flyweight or deliver an existing one.

## The Proxy Pattern

The intent of a *normal* proxy as defined in [8] is to provide a surrogate or placeholder for another object to control access to it. Dynamic proxies, on which our reflective model is based, play the same role with additional advantages of less coupling and a higher degree of code reuse. The type and the relationship of a normal proxy to target objects are fixed at compile-time. Those of a dynamic proxy are determined first at runtime. Dynamic proxies allow, using polymorphism and reflection, for implementing generic behaviour independent of the targets' type.

### 5.3. Creational Patterns

Factory patterns provide an interface for creating an object or families of related objects, which, depending on runtime data, decides which instance of the several possible sub-classes or class hierarchies to return and returns one. Singleton ensures a class only has one instance and provides a global point to access it [8].

As observed by Norvig [29] and Sullivan [33], languages which allow overriding of the default creation method provide direct support for the Factory Method and Singleton patterns. Also, for languages in which classes are first-class entities, there is no need for abstract factories since classes themselves serve as factories. The behavioural reflection model supported by dynamic proxies does not allow redefinition of Java's `new()` method for object creation independently of Factory classes. Moreover, in the underlying object model, classes are not first-class. As a result, there are no benefits to employing proxies for implementing Factory Method, Abstract Factory and Singleton patterns.

The Builder pattern separates the construction of a complex object from its representation (type and content) so that the same construction process can create different representations. It is similar to Abstract Factory but objects are created in more than one step. As for the factory patterns, applying the proxies' reflection model has no apparent advantages when compared with the static implementation.

The Prototype pattern is used when it becomes more appropriate to create objects by cloning rather than by instantiating the class. Java's interface `Cloneable` provides an implementation of the pattern. Objects must implement this interface and override the `clone()` method of the root class `java.lang.Object` before they can be copied. We note that `clone()` performs a *shallow copy* of the object. That is, the copy points to the same objects the copied object points to. The methods copies values of the fields in the new object, not the actual objects (references) the original object points to. If a shallow copy of the data is sufficient, the Java implementation of Prototype is simple.

## 6. Discussion

OOD is concerned with developing a model mapping the problem domain into classes and objects to implement the system requirements. Design activities and artefacts serve to document system structures and to provide a smooth transition in the implementation phase. There are no prepared recipes for building high quality software which is reusable, extendable, adaptable and reliable; but there are design principles, guidelines and concepts. Applying these principles and making use of the concepts enhance the quality of software. For example, following the idea of *Design by Contract* [27] enhances the reliability of the system. From the perspective viewing the relationship between a class and its clients as a formal agreement, the *correctness* of software elements can be addressed [26]. Also, applying the design principle of open implementation makes software systems more adaptive to changes. The main features of open architectures (systems that follow the principle of open implementation) are less coupling between components/objects and separation of concerns.

The proposed reflection model is not restricted to the Java environment and can be realised on other platforms supporting reflection, for example, Microsoft .NET. Dynamic proxies are runtime objects that conform to particular interfaces and dispatch all invocations to a generic handler. The `System.Reflection.Emit` Namespace of C# [28] can be used to generate a new class at runtime that implements required interfaces and passes control to a generic handler.

The flexibility gained by employing the behavioural reflection model is not without cost. The extra overhead imposed by reflection causes performance problems. One way to counter reflection effects is caching the proxy objects implementing reflective code (using other proxies). The problem of setting a caching policy or any other strategy to improve performance is left for future research.

The proposed reflection model based on dynamic proxies has parallels with the AOP approach; it exhibits a certain degree of separation of concerns. Applications at the base level can be identified with core concerns and the meta-program with crosscutting concerns (or aspects). However, separation of concerns in AOP does not necessarily mean weak (or loose) coupling; there is nothing that prevents aspects implemented in AspectJ (the most widely used language) [3] to explicitly refer to classes of the core concerns. In addition, core concerns and crosscutting aspects are combined at compile-time. Nevertheless, AspectJ allows structural change of application classes and provides a powerful join-points model which allows grouping of pointcuts. Pointcuts are program execution points at which aspect code is inserted (weaved into) at compile-time.

Table 1 summarizes the impact of applying the proposed reflection model on design patterns in terms of prominent features. The features include the usefulness of applying the model, whether applying the model simplifies the pattern implementation, promotes loose coupling and increases the level of code reuse as well as modularity. Applying the model is not useful if it complicates the implementation without apparent benefits. The conclusions related to loose coupling and code reuse are derived *intuitively* from class diagrams and coding examples.

Most pattern implementations resulted from applying the model show a higher level of modularity, due to the fact that the proposed model supports the separation of application objects from their meta-representations. Here, a module is conceived as a programming unit which handles one aspect of the problem being solved. In OO, a module is identified more closely with a class [16]. We note that some pattern implementations do not lead to autonomous and self-contained modules independent of applications specifics. For example, the Memento pattern implementation separates application objects from (meta-) objects manipulating their states; the meta-program, however, depends on the types of application objects (See Figure 24).

Empirical investigations are required to support or refute the claims about coupling and reuse features exhibited by the pattern We accept that we have not supported our claims with software metrics. However, case studies in previous works by the authors have indicated that reflective systems exhibit less coupling and higher level of code reuse when compared with identical non-reflective systems [14, 15].

For the Template Method pattern, the reflection model substitutes inheritance with object composition. For Strategy and State, applying the model simplifies pattern implementation. For Iterator and Prototype (creational pattern), Java provides appropriate abstractions as library classes that allow for a direct implementation. Applying the reflections model to the Observer pattern using JavaBeans classes allows for a generic implementation of event-trigger mechanism, thus increasing the level of code reuse. For Adapter and Decorator, the model provides a more flexible implementation; the adaptee is added through a proxy and the default implementation is kept as is. For some patterns, applying the model promotes loose coupling arising from the movement of some participants to the meta-level. For Interpreter and Flyweight patterns, applying the model complicates the implementation and worsens the performance. The reflection model is not useful when applied to creational patterns, because the model cannot be utilized to redefine the `new()` method.

| **Pattern's name** | **Prominent features of applying the reflection model** |
|---|---|
| Template Method | Substitutes inheritance by object composition |
| Strategy | Simplifies pattern implementation |
| State | Similar to Strategy pattern |
| Iterator | Not useful. Java's abstractions are satisfactory |
| Visitor | Promotes loose coupling |
| Observer | Using JavaBeans, increases level of code reuse |
| Command | Increases level of modularity & promotes loose coupling |
| Chain of Responsibility | Similar to Command pattern |
| Mediator | Promotes loose coupling further |
| Memento | Supports separation of concerns |
| Interpreter | Not useful |
| Adapter | Provides a more flexible implementation |
| Decorator | Similar to Adapter |
| Bridge | Useful in case open implementation feature is required |
| Composite | Promotes loose coupling and code reuse |
| Facade | Promotes loose coupling and code reuse |
| Flyweight | Not useful, due to costs of reflection overhead |
| Proxy | Not applicable |
| Cerational patterns | Not useful, because model cannot be utilized to redefine `new()` |

**Table 1. Features of design patterns when adapted to the reflection model**

## 7. Conclusions and future work

In this paper, we have provided an analysis of the reflective abilities of the Java language. The proposed reflection model established with the help of proxies provides a means of customising applications' behaviour and of adapting to new requirements without the need for modification. The reflection model supports application customisation in a way similar to the open implementation approach. Application objects provide a default representation, which acts upon and deals with some part of the real world (*problem domain*). The base representation is *closed*, meaning that it remains executable without being *merged* with the meta-representation. The latter representation provided is independent of applications; it is defined solely in terms of the service it provides. A meta-level architecture inherently exhibits a degree of separation of concerns, where the application core concerns (at the base level) are separated from the non-functional concerns (at the meta-level) in a natural way.

Extending the reflective architecture to incorporate the client/server model is straightforward and varies only according to the networking technology used to implement the communication protocol. In the client/server model, meta-objects providing customisation services become remote objects, running on a different JVM (on the server side) as their clients at the base level. We presented an example on tracing and showed how the single JVM application could be extended to a RMI based distributed system. In general, adapting a single JVM application to a distributed environment (with multiple JVMs) requires a definition of the interface between clients and server and a decision on which objects should run on the server JVM. In our case, it is clear that the meta-objects should be part of the server process and base objects client applications. In the client/server model, meta-objects represent remote services, accessible through remote objects. The interface between client and server is built on top of communication interface provided by the applied technology.

Applying the reflection model to design patterns showed that some pattern implementations become simpler. For a number of patterns the implementation exhibited less coupling, a higher degree of modularity and greater reuse. However, for many patterns, in particular creational patterns, the reflection model showed no apparent benefits over the traditional implementation.

In terms of future work, we are interested in investigating the question of applying the proposed reflection model to the Microsoft .NET platform. We are also interested in the question of using compile-time MOPs to implement reusable code and in examining the role of different programming languages in the implementation of design patterns.

## References

[1] Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., and Angel, S., 1977. A Pattern Language, Oxford University Press, NY.
[2] Aspect Oriented Software Development (AOSD) home page at: http://www.aosd.net/
[3] AspectJ home page at: http://www.eclipse.org/aspectj/
[4] Bobrow, D. G., DeMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., and Moon, D. A., 1988. Common Lisp Object System Specification, ACM SIGPLAN Notices, 23, 1-143.
[5] David, P. C., and Ledoux, T., 2002. Dynamic Adaptation of Non-Functional Concerns. In: Proceedings ECOOP '02 Workshop on Unanticipated Software Engineering, Malaga, Spain.
[6] Demers, F. N., and Malenfant, J., 1995. Reflection in logic, functional and object-oriented programming: a Short Comparative Study. In: Proceedings of the IJCAI '95 Workshop on Reflection and Metalevel Architectures and Their Applications in AI, Montreal, Quebec, Canada.
[7] Ferber, J., 1989. Computational Reflection in Class Based Object-Oriented Languages. In: Proceedings OOPSLA '89, SIGPLAN Notices, ACM Press, 317-326.
[8] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., 1995. Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA.
[9] Hannemann, J., and Kiczales, G., 2002. Design Pattern Implementation in Java and AspectJ. In: Proceedings OOPSLA '02, Seattle, WA.
[10] Hassoun, Y. and Constantinides, C. A., 2003. The development of generic definitions of hyperslice packages in Hyper/J. In: Proceedings ETAPS '03 Workshop on Software Composition, Electronic Notes in Theoretical Computer Science 82, No. 5.
[11] Hassoun, Y., Johnson, R., and Counsell, S., 2003. Reusability, Open Implementation and Java's Dynamic Proxies. In: Proceedings PPPJ '03, ACM International Conference Proceeding Series.
[12] Hassoun, Y., Johnson, R., and Counsell, S., 2004. Applications of Dynamic Proxies in Distributed Environments, to appear in the Journal of Software-Practice and Experience.
[13] Hassoun, Y., Johnson, R., and Counsell, S., 2004. A Dynamic Runtime Coupling Metric for Meta-Level Architectures. In: Proceedings CSMR '04, IEEE Computer Society Press.
[14] Hassoun, Y., Johnson, R., and Counsell, S., 2004. Empirical Validation of a Dynamic Coupling Metric. Birkbeck Computer Science Tech. Report, BBKCS-04-03, March 2004.
[15] Hassoun, Y., Johnson, R., and Counsell, S., 2004.Code Reuse Through Reflection: An Empirical Perspective, Birkbeck Computer Science Tech. Report, BBKCS-04-06, June 2004.
[16] Henderson-Sellers, B., 1992, A Book of Object-Oriented Knowledge: Object-Oriented Analysis, Design and Implementation, A New Approach to Software Engineering, Prentice Hall Object-Oriented Series.
[17] Java 2 Platform available at Sun Microsystems site at http://java.sun.com
[18] JavaSoft, CORBA Technology and the Java$^{TM}$ 2 Platform, Standard Edition, available at http://java.sun.com/j2se/1.4.2/docs/guide/corba/index.html
[19] JavaSoft, Java$^{TM}$ Remote Method Invocation, available at http://java.sun.com/products/jdk/rmi/
[20] Kiczales, G., des Rivieres, J., and Bobrow, D. G., 1991.The Art of the Metaobject Protocol, The MIT Press.
[21] Kiczcales, G., 1996. Beyond the black box: open implementation, IEEE Software, 13, 8-11.

[22] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J. M., and Irwin, J., 1997. Aspect -Oriented Programming. In: Proceedings ECOOP '97, Lecture Notes in Computer Science, 1241, 220-241.
[23] Kiczales, G., Lamping, J., Lopes, C. V., Maeda, C., Mendhekar, A., and Murphy, G., 1997. Open Implementation Design Guidelines. In: Proceedings of the 19th international conference on Software Engineering, Boston, MA, USA, 1997. ACM Press.
[24] Maes, P., 1987. Concepts and Experiments in Computational Reflection. In: Proceedings OOPSLA '87, ACM SIGPLAN Notices, 22, 147-155.
[25] Marcos, C., Campo M., and Pirotte, A., 1999. Reifying Design Patterns as Metalevel Constructs, Electronic Journal of SADIO, 2, 17-29.
[26] Meyer, B., 1997, Object-Oriented Software Construction, 2nd Edition, Prentice-Hall, Inc, NJ, USA.
[27] Meyer, B., 1991, Eiffel: The Language, Prentice-Hall, Inc, NJ, USA.
[28] Microsoft Visual C# .NET : Language Reference, 2002. Microsoft Corporation, Microsoft Press, Edmond, WA, USA.
[29] Norvig, P., 1996. Design Patterns in Dynamic Programming. In: Object World 96, Boston, MA.
[30] Object Management Group (OMG), Catalog of OMG Modelling and Metadata Specifications, available at http://www.omg.org/technology/documents/formal/uml.htm
[31] Parnas, D. L., 1972, On the Criteria To Be Used in Decomposing Systems into Modules, Communications of the ACM, 15(12): 1053–1058, December 1972.
[32] Smith, B. C., 1985. Reflection and Semantics in a Procedural Language, Ph.D.Thesis, MIT Laboratory of Computer Science, MIT.
[33] Sullivan, G. T., 2002. Advanced Programming Language Features for Executable Design Patterns, Lab Memo, MIT Artificial Intelligence Laboratory, number AIM-2002-005, MIT.