Using Aspects and Annotations to Separate Application Code from Design Patterns

Rosario Giunta  Giuseppe Pappalardo  Emiliano Tramontana
Dipartimento di Matematica e Informatica
Università di Catania
{giunta, pappalardo, tramontana}@dmi.unict.it

ABSTRACT

Design patterns are an invaluable resource to generate effective design solutions. However, employing some of them can be cumbersome for some application classes when these are forced to mix domain- with pattern-related code. Indeed, for a number of design patterns, we view the code implementing the behaviour required by their roles as a snippet, or concern, which is not involved in the domain issues that a class is supposed to address.

This paper proposes a methodology to obtain the behaviour described by some well-known design patterns, which allows better separation between domain and non-domain concerns through recourse to aspects and annotations. Indeed, application classes retain all the benefits that the design patterns are supposed to give them, while staying thoroughly separated from non-domain concerns. Moreover, fewer lines of code are needed to obtain a design pattern than in other approaches, and the propagation of changes to application code, caused by the introduction or removal of a design pattern, are greatly reduced.

Categories and Subject Descriptors

D.2.3 [Software Engineering]: Coding Tools and Techniques; D.3.3 [Programming Languages]: Language Constructs and Features

General Terms

Aspect-Oriented Programming, Design

Keywords

separation of concerns, design patterns, annotations

1. INTRODUCTION

Design patterns are solutions that have proved successful for recurrent problems and are usefully adopted for the sake of the high degree of modularity they bring about. However, usually the classical solution of a design pattern, as described e.g. in [6], leads to some application classes mixing code addressing domain-related concerns with code implementing the relationships among design pattern classes. We call such snippets of code non-domain concerns.

In this paper we propose, for the implementation of a few design patterns, some variations based on aspect-oriented programming [9] and meta-data, (i.e. Java annotations [13]), aimed to facilitate coding and the separation of different concerns into distinct entities. The proposed variations use the annotation support of the 1.5 version of Java [13] and the aspect-oriented language AspectJ [1].

Some aspect-oriented versions of design patterns have been proposed in [8, 7, 3, 11] to separate application code and design patterns which are encapsulated as aspects. For this purpose, however, [7, 3, 11] propose extending AspectJ with new, more powerful constructs, thus forcing the use of ad-hoc weavers or preprocessors. Moreover, all of the approaches [8, 7, 3, 11] require programmers to implement aspects as specialisations of suitable abstract aspects, as available, e.g., in [1]. As a result, a new “concrete” aspect, often consisting of several lines of code, has to be implemented, in order to make an application class play a design pattern role, while the latter class will remain completely unchanged. In contrast, we feel that new code to be introduced for this sake should be kept to a minimum, but (unlike the aspect) should leave a clear, albeit tiny, “mark” within the code of the class intended to take on the design pattern role. This helps improving the readability of such classes, by making their behaviour more explicit. A detailed comparison of our proposal with related work is provided in Section 6.

In keeping with the aspect orientation philosophy, we represent non-domain concerns as aspects, so as to isolate them from the functionality of classes. Moreover, we use annotations to make aspects independent of classes, hence completely reusable “as they are” for different applications. We employ annotations as a specification of how to connect classes to aspects, which we exploit to impose on applications classes a certain behaviour, conforming to an intended design pattern. The annotation hides the details of the code that will intervene at runtime in the form of aspects, but at the same time unobtrusively marks the class as one that plays a role for a design pattern. Making such a relationship explicit indeed renders application code more understandable, as deemed useful, e.g., by [10].

The design patterns we focus on, to illustrate our proposal, are Observer [6], Singleton [6], Proxy [6], and the static factory method [6]. Not all of the classic design patterns described in [6] can benefit from our approach. Indeed, design pat-
terns such as Façade, Mediator and State, while useful to suggest how to structure application code, do not require non-domain snippets to be introduced into code, hence do not present heterogeneous concerns that can be separated in the fashion we propose here.

In essence, our approach allows a Java application to be structured according to some fundamental design patterns and yet keep the impact on domain code to a minimum, thanks to the combined use of aspects and annotations. This is certainly useful in the development of a new software system, since it allows the programmer to readily exploit the solutions represented by design patterns. Furthermore, our methodology is also apt to encourage reusing existing application code: often, well-designed application classes focusing on a certain problem domain may be available for reuse, but will need to be integrated into a new software system in a variety of ways. However, since much of this variety can be catered for by established design patterns, such as Observer or Proxy, integration of existing classes conforming to the desired pattern is easily achieved through our methodology, simply by marking such classes with annotations that will subsequently drive the necessary aspect weaving. This essentially keeps unaltered the annotated domain classes to be reused.

Section 2, 3, 4 and 5 respectively show how the proposed methodology can be applied to the standard Observer, Singleton, Proxy design patterns and static factory method. Section 6 describes the existing related approaches and compares them with our proposal. Some conclusions are drawn in Section 7.

2. THE OBSERVER DESIGN PATTERN

2.1 Classical Solution

The intent of the Observer design pattern is to allow loose coupling between an object handling some data, and other objects interested to be notified when such data change.

The solution suggested by this pattern consists of roles Subject, Observer, ConcreteSubject and ConcreteObserver. ConcreteObserver is an application class interested to be notified when the state of application class ConcreteSubject changes. Subject handles a list of Observers and is responsible for notifying these when the state of its subclass (i.e. a class acting as a ConcreteSubject) changes.

Loose coupling between a ConcreteSubject and ConcreteObservers is essentially obtained because the ConcreteSubject delegates to its superclass Subject the handling of ConcreteObservers, and by making Subject depend on the interface Observer that ConcreteObservers have to implement. As far as ConcreteObservers are concerned, they are aware of Observer and Subject.

2.2 The Aspect-Oriented Version of [8]

In the classical implementation of the Observer pattern, ConcreteSubject has to invoke Subject methods when its state changes, so as to notify ConcreteObservers. For this, the code within ConcreteSubject, besides addressing some functional concern, has to be augmented with the non-domain code that invokes the Subject methods.

In the aspect-oriented version [8] of Observer, an aspect is responsible for: (i) intercepting the relevant methods of each class playing the role of ConcreteSubject, i.e. the methods that change the state of the ConcreteSubject, and (ii), after their execution, notifying the interested ConcreteObservers.

Thanks to these “observing” aspects, ConcreteSubject need not be aware of Subject and is not responsible for invoking Subject methods. It is the “observing” aspect that contains the list of methods of ConcreteSubject that have to be intercepted. In this way, the design pattern related code is moved out of the ConcreteSubject class and only addressed by an aspect. Moreover, ConcreteObservers need not be aware of any classes of the design pattern Observer, since they are forced by the aspect to unknowingly implement the appropriate interface.

2.3 Discussion

In the aspect-oriented solution presented, the “observing” aspect must contain the list of methods that have to be intercepted, thus a change in the ConcreteSubject’s interface, e.g. a method renamed, would affect the aspect code. Of course, a general abstract aspect can be used to implement all the code that is independent of the application’s ConcreteSubjects and ConcreteObservers. However, as in [8], one or more aspects inheriting the abstract one will have to be implemented in order to connect any new ConcreteSubject with the relevant ConcreteObserver.

As a result, keeping the related aspects updated when renaming a method of a class acting as a ConcreteSubject is an activity that programmers could accidentally forget, and represents a little, undesirable burden anyway. Moreover, no message would come from the compiler if a dependence were broken due to method renaming.

In some circumstances, the “observing” aspect might include the list of methods of more than one ConcreteSubject. This would increase the aspect’s degree of dependence on application classes, possibly exacerbating the said problem.

2.4 Aspect-Oriented Version with Annotations

We propose to use an annotation, i.e. @Observer, to relate the state changing methods of a ConcreteSubject to the aspect responsible for implementing the Observer design pattern. Using the @Observer annotation, the aspect is independent of ConcreteSubject’s methods, thus staying unaffected when these change. Moreover, the aspect works for any number of ConcreteSubjects and, of course, is reusable should new ConcreteSubjects be introduced.

The @Observer annotation gives an application class the behaviour that the Observer design pattern is intended to provide, without requiring the related code to be introduced into the class. This annotation is used as a “modifier” of the method of a class and provides the (string) names of: the class to be notified, i.e. the ConcreteObserver, the method to be called, and the parameter taken by such a method (see Figure 1).

```java
@Retention(RetentionPolicy.RUNTIME)
public @interface Observer {
  String clas();
  String meth();
  String par();
}
```

Figure 1: Definition of an annotation @Observer for the Observer design pattern

Figure 2 shows an example of use of annotation @Observer. Class Account plays the role of a ConcreteSubject as annota-
tion @Observer(class="Store",meth="update",par="bal") marks method deposit(). Such an annotation tells that method update() of class Store should be called with parameter bal after method deposit() returns. Hence, class Store plays the role of a ConcreteObserver. The target of annotation @Observer is a method, so as to indicate which operations of the class change the observed state.

```java
public class Account {
    private double balance = 0;
    @Observer(class="Store",meth="update",par="balance")
    public void deposit(double sum) {
        balance += sum;
    }
}
```

Figure 2: The @Observer annotation used within class Account

As listed in Figure 3, the aspect acts as a connector between the annotated method and the ConcreteObservers. Upon interception of annotation @Observer, by pointcut observer, execution context is gathered, which consists of instance obj of the annotated class and that of annotation ann. Then, the advice: (i) reads the arguments of annotation ann, (ii) obtains a representation of the class and method of the ConcreteObserver, as c and m, respectively, (iii) from hashtable subjects, held by the aspect, looks up list observers corresponding to instance obj, (iv) retrieves the observed parameter and its value, as par and pp, respectively, and finally, (v) calls method m, employing dynamic invocation \[5\], for all the interested instances in list observers.

```java
public aspect ObserverPattern {
    private Hashtable<Object, List<Observer>> subjects = new Hashtable<Object, List>();

    //capture methods annotated with @Observer
    pointcut observer(Observer ann, Object obj): this(obj) &&
    execution(@Observer * *(..)) && @annotation(ann);

    after (Observer ann, Object obj): observer(ann, obj) { try {
        //get class and method of observers
        Class c = Class.forName(ann.class);
        Method m = c.getMethod(ann.meth(),
                              new Class[0](Object.class));
        //get list of observers instances for the subject
        List observers = subjects.get(obj);
        //get the parameter of the subject
        Field par = obj.getClass().getDeclaredField(ann.par());
        Object pp = par.get(obj);
        //invoke all observers
        if (observers != null) for (int i=0; i<observers.size(); i++)
            if (observers.get(i).getClass() == c)
                m.invoke(observers.get(i), pp);
        } catch (Exception e) { /* ... */ }
    }

    public void addObserver(Object subj, Object obs) {
        List<Object> obsList = subjects.get(subj);
        if (obsList == null) obsList = new LinkedList<Object>();
        obsList.add(obs);
        subjects.put(subj, obsList);
    }
}
```

Figure 3: Aspect implementing the “observing” concern

When the method of the ConcreteObserver to be called by the aspect (e.g. update(), in the example of figure 2) is not a static one, the references to the instances that need notification have to be provided. This is possible thanks to method addObserver() provided by the “observing” aspect ObserverPattern (see Figure 3). Such a method has to be invoked, as ObserverPattern.aspectOf().addObserve(…), passing the instances of application classes playing roles ConcreteSubject and ConcreteObserver, respectively. Similarly to addObserver(), another method has been provided for allowing a ConcreteObserver to be removed, at runtime, from the list of objects that need notification.

In the proposed version of Observer, aspect ObserverPattern uses introspection and dynamic invocations (see Figure 3) to be independent of application classes and hence reusable as is. While such Java features could be unknown to the average application programmer, the latter need not understand the pattern’s inner workings anymore than that of many Java standard libraries. Indeed, aspect ObserverPattern and annotation @Observer could even be introduced as a library support for aspect-oriented languages.

For the application developer to use the suggested version of Observer, she has only to annotate the methods of class ConcreteSubject changing the state of the instance. Unlike the classical solution, this version need not make calls to the superclass and therefore the code of its methods does not intertwine different concerns. Moreover, application classes playing roles ConcreteSubject and ConcreteObserver are not constrained by hierarchical relations. Within the code of ConcreteSubject, the annotation holds information about its ConcreteObserver, this makes it explicit the correspondence between two application classes, without code tangled (i.e. method calls or hierarchical relationships). For removing such a correspondence, it suffices to remove the annotation that will be easy to spot since located separately from the code implementing the method.

An interesting side effect of our version of Observer is that different methods of the same ConcreteSubject can be independently observed by instances of distinct classes. This can be useful e.g. when operations change different parts of the observed state, or when the same state can be changed in several ways, and the observers are just interested in some part of the state or in some of the possible changes. If needed, annotation @Observer marking a method can be given arguments that are not as those of the annotation marking another method. This would allow alerting different methods and possibly different classes when some operations of ConcreteSubject are executed. In contrast, the classical approach forces all ConcreteObservers to be notified for all operations changing the state of a ConcreteSubject.

Figure 3 shows the version of aspect ObserverPattern where only instances of one class playing the role of ConcreteObserver have to be notified for each method. However, an extended version handling a list of ConcreteObservers, using annotation @Observer(class=\"View\",\"Store\",\ldots) can be easily implemented and has not been shown for the sake of brevity.

3. THE SINGLETON DESIGN PATTERN

3.1 Classical Solution

The intent of the Singleton design pattern is to force only a single instance to be created for a class at runtime and to
provide an access point for such an instance.

The proposed solution is to introduce, for the Singleton class, a private constructor, so that instantiation using new cannot be performed outside the said class. A static method, e.g. getInstance() is provided to return the unique instance of the class, held by a static attribute, and check whether an instance has been already created, in that case avoiding to create further instances. Note that no application classes can use new on a Singleton class, but should instead resort to getInstance(), which is a static method, as the access point to the class.

The non-domain code for the Singleton pattern consists of the private constructor, the static method (e.g. getInstance()) returning the unique instance, and the static attribute holding the instance.

Thus, changing an application class to make it a Singleton, once it has been implemented, triggers some changes to other application classes that originally needed to instantiate it. These classes have to be changed in order to use the newly implemented static method getInstance() in lieu of instantiation via new.

### 3.2 Aspect-Oriented Version with Annotations

In the proposed solution, annotation @Singleton (see figure 4) marks the class that should become a Singleton, and an aspect captures the instantiation of such a marked class, taking care to instantiate it only once. The annotated class needs no other editing, i.e. the constructor need not be changed and no static method returning the instance is necessary. More importantly, once a class has been made a Singleton using the annotation, no changes need to be propagated to application classes.

```java
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
public @interface Singleton {
}
```

Figure 4: Annotation for the Singleton design pattern

As an example, for class Bank to behave as a Singleton, it only has to be annotated by @Singleton public class Bank[...], and no other application class has to be changed.

Unlike the classical solution, the overhead for developing the proposed solution for the Singleton behaviour is minimal. Note that, should the intended Singleton class have more than one constructor, the classical solution would force all constructors to become private, and a getInstance() method to be implemented for each such constructor. This is unnecessary in our approach. Moreover, existing application classes that invoke new on a Singleton class need not be changed, for the aspect will take care of providing them with the unique instance needed. Hence, with our approach there is no propagation of changes when converting a class into a Singleton.

Listing 5 shows the aspect implementing support for the Singleton behaviour. It will intercept all the instantiations of classes marked by annotation @Singleton, obtain the class of the captured instance, and check whether an instance already exists for it by looking it up on table singles holding class-instance pairs. When such an instance exists, it will be returned, instead of creating a new one. Otherwise, the aspect lets the instance be created (by using proceed()); the proper constructor executed, table singles updated, and then returns the created instance.

```java
import java.util.Hashtable;
import java.lang.annotation.*;
public aspect SingletonPattern {
    private Hashtable<Class, Object> singles =
        new Hashtable<Class, Object>();
    // capture instantiation of annotated classes
    pointcut constr (): call ((@Singleton +).new(..)) &&
                     !within(SingletonPattern);
    Object around(): constr () {
        Object obj = null;
        Class s = thisJoinPoint.getSignature().getDeclaringType();
        if (singles.get(s) != null) {
            obj = proceed();
            singles.put(s, (Object) obj);
        } else
            obj = singles.get(s);
        return obj;
    }
}
```

Figure 5: Aspect implementing the Singleton concern

#### 4. THE PROXY DESIGN PATTERN

##### 4.1 Classical Solution

The intent of the Proxy design pattern is to have a surrogate object that controls accesses to a target object.

In the proposed solution, an interface, named Subject, is defined and two classes RealSubject and Proxy implement such an interface. Invocations to methods of RealSubject are performed by Proxy, which is the only one holding the reference to the target object, i.e. RealSubject. All other application classes should then access RealSubject only through the class playing the role of Proxy.

##### 4.2 Aspect-Oriented Version with Annotations

For an application to use the Proxy design pattern in keeping with the proposed methodology, two classes have to be implemented, so as to play the roles of RealSubject and Proxy. In contrast with the classical solution, application classes directly invoke the class acting as RealSubject, which must only have been marked with annotation @Proxy (see Figure 6). The annotation also carries a parameter specifying which class will be a Proxy for the annotated RealSubject.

```java
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
public @interface Proxy {
    String clas ();
}
```

Figure 6: Annotation for the Proxy design pattern

Aspect ProxyPattern (see Figure 7) is used to intercept annotation @Proxy, and re-direct execution to the namesake method of Proxy. In more detail, upon interception of an invocation to operations of an object whose class is marked @Proxy, by pointcut req, context is gathered, which consists
of references to called object o, invoking object t and annotation a marking the called object. The related advice checks whether the name of the class invoking the captured operation, is the same as the name of the class in annotation a. If so, then the call has been performed from the proxy and therefore execution is allowed (using proceed()). Otherwise, dynamic invocation is performed on the namesake method of the one intercepted. For such an invocation, a representation of the method, m, is needed, which is built by using the name of the method intercepted, ms, proxy class c, and the parameters types as in the intercepted call. The reference to the proxy corresponding to the invoked object o is retrieved from list proxies, which is an attribute of the aspect, holding all the instance pairs for a RealSubject and its Proxy.

Within the same aspect, another pointcut, creation, intercepts the instantiation of classes annotated with @Proxy. Such a pointcut gathers the reference to the invoking object, as t. The corresponding advice reads annotation @Proxy on the class to be instantiated and checks whether the class of the captured instance is the same as that of the annotation. If so, the instantiation is being performed by the proxy, then proxy is provided with the reference held by tmp within the aspect. Otherwise, the instantiation is allowed (using proceed()), a proxy is instantiated, and the proxies list is updated.

Class Proxy must implement the logic that decides whether access to RealSubject is allowed. As an example, in order to introduce a CheckAccount proxy controlling access to class Account, so that the latter behaves as a RealSubject, we only need to annotate Account by @Proxy(class="CheckAccount").

In our approach, an application class using another class that plays role RealSubject is unaware that access to it will take place through a proxy. Hence, introducing or removing a proxy during design time does not imply any change to application classes, other than simply annotating the class RealSubject. Comparing our solution with the classical one, it is possible to note that with our approach we obtain a substantial reduction in the amount of changes required in evolving a design.

In the proposed version of Proxy, application classes need not be changed to deal with a proxy class, i.e. no changes of types or instantiations. Moreover, once a class has been annotated by @Proxy, its instances will be always shielded from direct access. This prevents developers dealing inconsistently with access rules for a class, however no direct access can be granted to selected classes. In order to provide such an access the developer has to customise aspect ProxyPattern implementing the code that grants some application classes direct access to RealSubject. Such a code simply consists of the statement &k & !within(anAppClass) to be added to both the pointcuts of ProxyPattern, where anAppClass is the class allowed to have direct access to RealSubject.

5. THE STATIC FACTORY METHOD

5.1 Classical Solution

Similarly to the Factory Method design pattern, a static factory method is one that encapsulates the logic needed for creating instances, but, unlike the pattern, simply represents a method within the class to be instantiated. A static factory method allows proper handling of reference objects, i.e. objects that represent some real-world objects and whose instances should be retrieved when needed, rather than creating a new instance again when needed (as for a date).

While the Factory Method design pattern in [6] specifies roles to be played by application classes, it does not require these to be interspersed with new non-domain code, hence does not lend itself to our approach. This is not so for the static factory method [4], as discussed below.

5.2 Aspect-Oriented Version with Annotations

In the proposed solution, three marker annotations are used. Annotation @Factory (see Figure 8) is used to mark the class whose instances are reference objects. Hence, their references need to be stored and then, when needed, retrieved according to a given key. Moreover, annotation @Accessor is used to mark the value that is passed as a parameter of the constructor and that will be used as a key to retrieve the reference to the object. Finally, annotation @FactoryMethod is used to mark the method that returns the existing instance given a corresponding key. This
method need not be implemented, the necessary logic is provided as aspect code.

```java
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Factory {
}
```

Figure 8: Annotation indicating that a class is a Factory

In the static factory method, the non-domain code deals with managing, storing and accessing the representations of reference objects. It is implemented by aspect FactoryPattern (see Figure 9), thus separating it from domain classes.

```java
public aspect FactoryPattern {
    private static Dictionary<Object, Object> inst = new Hashtable<Object, Object>();

    //capture instantiations of the class annotated with @Factory pointcut constr(): call((@Factory *) .new(..)) && !within(FactoryPattern);
    Object around(): constr() {
        //for the class annotated with @Factory, extract annotations of the parameters of its constructor
        ConstructorSignature s = (ConstructorSignature) thisJoinPoint.getSignature();
        Annotation[][] an = s.getConstructor().getParameterAnnotations();

        //extract the position of parameter annotated with @Accessor
        int keyArg = -1;
        for (int r = 0; r < an.length; r++) {
            for (int c = 0; c < an[r].length; c++)
                if (an[r][c].annotationType().getName().compareTo("Accessor") == 0) {
                    keyArg = r;
                    break;
                }
        }
        if (keyArg > -1) break;

        Object[] args = thisJoinPoint.getArgs();
        Object obj = proceed();

        //use the value of the parameter annotated with @Accessor
        //as key to access the new instance
        inst.put((Object) args[keyArg], (Object) obj);
        return obj;
    }

    //capture methods annotated with @FactoryMethod pointcut access(Object obj): !within(FactoryPattern) &&
    //call((@FactoryMethod *) .execute(*.*) && (Object)obj)
    //& execution(FactoryPattern && FactoryPattern *) (*(Object));

    //provide the reference corresponding to given key
    Object around(Object obj): access(obj) {
        return inst.get(obj);
    }
}
```

Figure 9: The Factory aspect

Aspect FactoryPattern captures the creation of instances of the class annotated by @Factory, and stores the key-reference pairs into hash table inst. The key is given by constructor parameter annotated as @Accessor. In the listing, firstly the annotations on the constructor arguments are obtained, secondly @Accessor marker is searched, so as to recognise the parameter used as a key for accessing the created instance, then the key-created instance pair is stored into inst. Finally, the aspect captures invocations to the method marked with @FactoryMethod, and returns references to the requested object (the reference is retrieved according to the key passed as parameter).

An application using the suggested version will be freed of the code needed to store and later access instances. Only application classes whose instances have to be handled as reference objects have to be marked by the said annotations.

6. RELATED WORK

Recently, AspectJ 5 has included support for annotations [1]. This permits an enhanced selection of join points, based on the additional attributes of the code made available by annotations. Moreover, annotations allow the design of aspects that are well separated from classes. Similarly, the JBoss [2] framework provides support for implementing aspects that select execution according to annotations.

Our proposed contribution is based on AspectJ 5’s ability to select execution inside aspects on the basis of annotations, which allows non-domain concerns embodied within aspects to be kept out of application classes, leaving the relevant code well-separated.

The non-domain concerns that we focus on regard the structure of the application and aim at shaping it in accordance with some well-known design patterns. Following our proposed approach, the resulting code of classes retains all the structural advantages afforded by (the classical versions of) design patterns, but is at the same time not forced to include non-domain code. This is clearly an advantage for the development and evolution of the system. Parametric aspects have been proposed to allow the implementation of an abstract aspect that may be employed to define the roles of a design pattern, the relationships between them and their behaviour [3]. The resulting aspect-oriented language would include extended constructs such as roles, generic, multiple, class-set and members, which make it possible to define the Abstract Factory pattern. The abstract aspect, which can be reused being independent of applications, has to be specialized by concrete aspects in order to bind application classes to the roles of the intended design pattern. Thus, at weaving time, an ad-hoc weaver can generate application classes that conform to the roles and behaviour defined by the abstract aspect for the given design pattern.

The idea of [3] of modelling pattern roles through abstract aspects is similar to the approach of [8], but is simpler to apply thanks to the ad-hoc, more expressive, constructs. There are several issues that make modelling of design patterns through parametric aspects cumbersome. Firstly, an ad-hoc weaver is required to generate code according to the specification given as an abstract aspect. Secondly, to define the role that a particular class plays for a design pattern, the programmer has to write a specific concrete aspect, which typically consists of several lines of code. Thirdly, the programmer might need to edit the generated code in order to make it complete. Finally, a set of constructs have to be introduced with the aim to specifically support the description of design pattern Abstract Factory, such as multiple and class-set. These constructs have no meaning for other design patterns. In order to extend the approach to other design patterns, it could be the case that yet more specific constructs are needed. This seems to negatively affect the balance of specific constructs that a general purpose lan-
language should exhibit.

In [7], the authors propose the concept of **parametric introductions**, a device which, like static crosscutting, lets programmers inject into classes code that depends on parameters that will be provided at weaving time. For this purpose an extended aspect-oriented language is needed. According to [7], for a class to behave in accordance with a design pattern, a suitable aspect must be provided to inject appropriate code into the class. E.g., for any class that should become a Singleton, a (parametric) aspect will take care of inserting into it a static method `getInstance()` and a static attribute (the type of both will be determined, via parametric introduction, at weaving time). Another aspect specialising the former will then be needed for each class supposed to become a Singleton. In general, as in [3] and in contrast with our proposal, a specific aspect has to be implemented to confer to an application class the role it should play in a design pattern. However, in the approach at hand, an extended syntax for the aspect-oriented language allows more powerful pointcuts to be defined, which results in an easier and more concise specialisation effort. A further difficulty is that, once the aspect has been introduced, other parts of the application are affected and still need to be manually changed for the desired design pattern to be actually applied. In the Singleton example, any new used in application code to instantiate the class turned into a Singleton will have to be replaced by a `getInstance()` call.

In [11], an extension of AspectJ is proposed that allows **logic variables** to be used in order to represent packages, types, fields and methods within “generic” aspects. A generic aspect specifies the binding between a logic variable and, e.g., a class by implementing some conditions. Such conditions evaluate whether the execution context, e.g. methods and classes with given names, are actually available. As an example, they obtain the Decorator design pattern in terms of a “generic” aspect that is reusable across applications. Still, specific aspects will be needed to define which application classes are to play a role for the design pattern.

## 7. CONCLUSIONS

This paper has proposed a different implementation of a few widespread design patterns. The proposed scheme uses annotations to mark a class (or part of it) which should behave according to a role of a design pattern. Moreover, aspects are used to encapsulate the design pattern concern, and to impose the intended role’s behaviour on annotated classes at runtime.

The proposed approach affords some clear and significant benefits: (i) application classes and aspects remain independent of each other; (ii) aspects are fully reusable “as they are” for any class to be conferred a new role (no specialisation is needed for the target class); no ad-hoc compilers or supporting libraries are required. Finally, we regard it as desirable that a tiny annotation is required to mark the role that a class will play within a design pattern: we deem that such these explicit information bits will make the code easier to understand.

The use of aspects that implement non-domain concern, as in the above approach, is general enough to be used for other aspect languages rather than AspectJ. However, our pointcuts rely on the ability to capture annotations. Without annotations, pointcuts would depend on application classes and their method signatures, hence would not be as general as to be freely reused across applications. To the best of our knowledge no language other than AspectJ provides the said support for annotations. Since aspect-oriented languages are still evolving, we expect such a support would be made available later on.

Currently, the AspectJ weaver is able to operate both on the source and bytecode of Java classes. However, our proposed Java annotations have to be inserted into the source code of an application. Further developments in weavers technologies could make it possible to have classes, provided as bytecode, and a separate description of markers, used as our annotations, that identify the class characteristics that are of interest for aspects. In such a way, the desired behaviour could be imposed to classes even when their source code would not be available. Spring framework [12] is an example of such a technology, whereby weaving is driven by an XML code, separated by both classes and aspects, specifying the mapping between classes, their methods and aspects.

## 8. REFERENCES