Università degli Studi di Catania
facoltà di scienze matematiche, fisiche e naturali
Dottorato di Ricerca in Informatica

XXIV ciclo

Experiments towards a General Implementation of some Design Patterns using Aspect Orientation

Rosario Graziano Giunta

Tesi di Dottorato

Tutor: Chiar.mo Prof. Giuseppe Pappalardo

Coordinatore Chiar.mo Prof. Domenico Cantone

Printed 2011/12

Abstract

This dissertation presents a novel technique to implement the behaviour of some widely used design patterns using a combination of aspect oriented programming and computational reflection.

Object-oriented languages do not support design patterns as a language construct, instead these have to be specifically implemented by the programmer. As a result, their implementation ends up scattered over different classes and tangled with the domain code of such classes, leading to reusability, modularity and comprehensibility issues.

The aspect-oriented implementations presented in this thesis get rid of such issues. A design pattern is implemented as an aspect which, by intercepting an annotation that marks an application class, enforces the role of the pattern on that class, thus making the pattern available for the system.

Such implementations enjoy four properties, especially defined from the analysis of the literature, considered useful and not found together in existing approaches.

Efficient variants of the proposed approach are also described and compared with the standard object-oriented approach in terms of running times.

The proposed implementations can be used in object-oriented legacy applications, applying specific refactoring steps to convert legacy code to make it use the aspect versions.

Contents

Chapter 1

Introduction

This dissertation puts forward a novel modularisation of several widely used Design Patterns [GHJV94, BMR+96] using Aspect Oriented Programming [KLM+97] and Computational Reflection [Mae87].

Several Aspect Oriented Design Patterns (AODPs) are implemented by means of completely reusable aspects to enhance the modularity of the application using them. The classes of the latter need not to be aware of a design pattern's role they might play, as an aspect takes care of the enforcing of the pattern behaviour.

An object-oriented design pattern describes a solution for a recurring design problem in terms of relationships and interactions between classes and objects of an object-oriented system, it is used as a known solution to be implemented, with known advantages and drawbacks. In Software Engineering a software product has to respect some fundamental properties such as robustness, correctness, maintainability and reusability [Som01, Pre05]. The use of object-oriented design patterns is very helpful in the design of a software system as it allows the software to be structured in such a way to anticipate its changes and improve maintainability and reusability of its components. As a software product is prone to changes during its life-cycle, design patterns are very useful for supporting different kinds of changes: to adapt the software to different environments (adaptive maintenance), to improve its internal structure (preventive maintenance) and to extend its functionalities (perfective maintenance). The benefits of using design patterns are widely acknowledged as paramount also when dealing with the refactoring of a major software application [FBB+99, Ker04].

A design pattern defines *roles* that an involved class can play, i.e. a set of software structures and behaviours it has to conform to. Often these roles impose additional code to be added to the related classes to allow them to play the specific roles. While this additional code allows the implementation of the design pattern in the system, thus bringing its benefits, it comes at the price of some significant drawbacks, as summarised in the following, that can be lifted using the approach proposed in this dissertation.

Every time a programmer has to implement the same design pattern for different applications, she has to write very similar code and usually will not be able to reuse previous implementations of the same pattern, as such implementations had been especially tailored to the classes on which they had been applied to.

Thus, a class implementing a role for a design pattern becomes longer, more complex and difficult to understand, in addition it will not always respect the Separation of Concerns [HL95] principle, as its code would address both its functional (or domain) responsibilities, and those related to the design pattern's role.

In a system, all the classes that interacts with the ones implementing a role will often have to be aware of the pattern implementation and thus become tightly coupled with it.

Removing the code implementing a role for a pattern from a class is not a trivial task, as there is no sharp separation in its code between the functional responsibilities and the role-related ones. Moreover, changes on the class might propagate to other classes of the system that interacted with the role-implementing class, as the coupling between classes has increased.

To illustrate these drawbacks it is useful to briefly discuss a simple and widely used design pattern: *Singleton*. This pattern is used to limit the instances of a class to just one. To do so, the involved class must not expose any public constructors, and any access to a Singleton class must pass through a static method which returns the only reference to the instance.

To make an existing class a *Singleton*, it has to be modified by making any constructor private and by adding the public static method (`getInstance()`) hosting the code to manage the only instance. Now, any client class accessing a Singleton can not use the constructor but has to use the `getInstance()` method. Thus, if (when) the *Singleton* class stops playing this role in a later phase of development, the `getInstance()` method should be removed and its constructors made public. As its public interface changes, this triggers all the client classes to be changed accordingly, i.e. the programmer has to modify them to use the now-exposed public constructor instead of the `getInstance()` method. Such propagation of changes negatively affects the maintainability of the code base, rendering the use of design patterns less effective than they were originally intended.

All of the summarised problems are known and the literature presents different approaches to solve them while preserving the design patterns' benefits. However, existing approaches lack a way to avoid some of the summarised drawbacks, as it is extensively covered in chapter 6.

As a tool to tackle these limitations of the object-oriented design patterns, several authors [NK01, HK02, HB02] advocate the use of aspect orientation as a way to a better modularisation, the most notable approach being the well-known work by Hannemann and Kiczales [HK02]. They partition the behaviour of a design pattern into abstract and concrete aspects, so as to reuse the abstract ones, which contains the basic pattern's behaviour, for any application, while the programmer has just to specialise the concrete ones with the actual application classes. However, since the concrete aspects ultimately depend on application classes and some other ad-hoc code, concrete aspects can not be fully reused, just some components can. Moreover, certain aspects impose undesirable limitations on the implemented design patterns (see chapter 6).

In this dissertation the aspect-oriented language used is AspectJ [hp11], a widespread extension for the Java programming language[1]. The AODPs proposed in this dissertation use an aspect to encapsulate the behaviour needed for a class to play a given role in a design pattern. Such an aspect is independent of application classes and thus completely reusable as is. The gluing code to impose an application class to perform a role for a design pattern is an annotation to be added to such a class. Once the aspect is woven into the application, the expected role behaviour for the class is automatically enforced by the aspect. To remove such a role from the application, it is sufficient to remove the annotation and compile again.

The proposed aspects can be generic thanks to the use of computational reflection, used at runtime to gain any additional information needed to perform the pattern-related tasks.

[1]Please note that the proposal is not tied to any specificity of the said languages and is in principle applicable to any sufficiently developed aspect-oriented language implementation for an object-oriented language.

The use of reflection causes longer running times, so two possible alternatives to the main solution are provided. One is the caching of some results of the reflective methods' calls and another is the generation of specialised aspects from an aspect template. All these versions provide the same black-box behaviour.

This dissertation is structured as follows. In section 1.1 the four essential properties of the proposed aspect-oriented approach are presented. Chapter 2 deals with the tools that allowed the AODP's implementations. Chapter 3 and 4 detail the proposed patterns' implementations in their three variants: reflective, cached and specialised. In chapter 5 an overall evaluation of the proposed implementations is presented, and such implementations are analysed in terms of running times against a regular object-oriented solution. Chapter 6 presents a literature review, with particular care for the comparison with the state-of-the-art approach of [HK02]. The author's conclusions are drawn in chapter 7.

1.1 Properties of aspect-oriented design patterns

The drawbacks of the object-oriented design patterns described in the previous section, in concert with a thorough analysis of the existing literature, have led to the definition of the following desired properties for a design pattern implementation. In particular, the AODPs proposed in this dissertation verify each one of these properties, while the previous approaches fail to satisfy at least some subset of these properties, as the literature review shows in chapter 6.

Separation of Concerns (SoC) – The functional (domain) code of a class should be completely separated from the code implementing a design pattern role. This improves the maintainability of the software system and reduces maintenance costs.

Entire Characterisation of Roles (ECoR) – The role of a design pattern should be fully specified in a single module, i.e. not spread into different components, in a generic way, i.e. not tied to any specific class. This makes it easy to understand and reuse it in different contexts (application) without any changes.

Single Point of Change (SPoC) – When a role has to be added to (or removed from) a class, there should be just a single part of the code to change, to minimise the propagation of changes.

Robust Enforcement of Roles (REoR) – A design pattern's role implementation should be general and robust enough to be used in different applications and contexts without changes, possibly disallowing a wrong use of the pattern.

In order to summarise the above desired characteristics: the code imposing a role from a design pattern to some classes of an application should be separated from the functional concerns of the classes involved (*SoC*), possibly fully specified (*ECoR*) in just one module, loosely coupled with other classes of the application (*SPoC*) and possibly capable of being used in any application without changes, assuring the programmer that the design pattern will work as expected in the application (*REoR*). These characteristics are true for the AODPs described in the following chapters.

The appropriate use of Aspect-Oriented Programming (AOP, see section 2.2) allows the implementation of these characteristics. The implementations of the roles for a design pattern are kept in a single module (an *aspect*) which hosts all the needed code (*SoC*, *SPoC*). The generality of these implementations is achieved by means of computational reflection (section 2.1), thus making an aspect (i.e. a design pattern) completely reusable (*ECoR*). Activation of the aspect is triggered by well-defined rules (pointcuts, see section 2.2) of the aspect, thus the programmer can not misuse the design pattern (*REoR*); moreover the superimposition of a role on a class is turned on by adding the aspect to the application, and removed by compiling the application without the aspect (*SPoC*).

1.2 Papers 2008–2011

During his PhD years the author of this dissertation has been a co-author of the following papers on aspect orientation and design patterns:

- *Using Aspects and Annotations to Separate Application Code from Design Patterns.* In the 25th ACM Symposium on Applied Computing (SAC 2010), *Programming for Separation of Concerns* track [GPT10];

- *Aspects and Annotations for Controlling the Roles Application Classes play for Design Patterns.* In the 18th Asia-Pacific Software Engineering Conference (APSEC 2011) [GPT11];

- *Superimposing Roles for Design Patterns into Application Classes by means of Aspects.* In the 27th ACM Symposium on Applied Computing (SAC 2012), *Programming for Separation of Concerns* track [GPT12b];

- *AODP: Refactoring Code to Provide Advanced Aspect-Oriented Modularization of Design Patterns.* In the 27th ACM Symposium on Applied Computing (SAC 2012), *Software Engineering* track [GPT12a].

He also co-authored the following papers on large-scale distributed systems:

- *Analysing the Performances of Grid Services Handling Job Submission.* In the 18th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE 2009) [GMPT09a];
- *Measuring Performances of Globus Toolkit middleware Services.* In the Final workshop of GRID projects "PON Ricerca 2000–2006, Avviso 1575" [GMPT09c];
- *Improving the Performances of a Grid Infrastructure by means of Replica Selection Policies.* In the Final workshop of GRID projects "PON Ricerca 2000–2006, Avviso 1575" [GMPT09b].

Chapter 2

Used tools

In order to understand and appreciate the proposed solutions it is useful to firstly introduce the tools used for their implementation. The following sections provide a basic introduction on these tools with particular care on the most important ones.

In section 2.1 an overview of Computational Reflection is presented, with particular emphasis on the standard Java API providing such support. Section 2.2 deals with the main concepts of Aspect Orientation, especially with the most used ones in the rest of this dissertation. Section 2.3 provides a basic summary of Java annotations.

2.1 Computational reflection

In [Mae87] Pattie Maes defines a reflective system as a "system which incorporates structures representing (aspects of) itself". A computational system bearing such a feature is able to answer messages about (aspects of) its internal structure(s), i.e. the system exposes an interface to access its internal structures, and thus can change its own behaviour.

One of the basic models for such a system is depicted in figure 2.1 (other models exist [Fer89]). In it, each object in the system can be paired with a metaobject, with the former unaware of being paired with the latter. Using this model, when a programming language supports *computational reflection* (or simply *reflection*), two different types of possible operations are allowed to the programmer: *introspection* and *interception*.

A metaobject (*mo*) is allowed to introspect the object (*o*) to which it is paired,

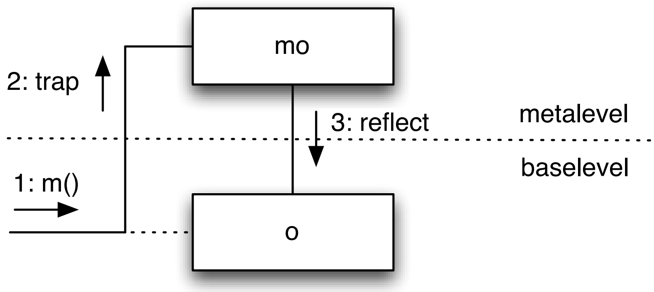

Figure 2.1: *Pairing of an object and a metaobject in a reflective system*

this allows *mo* to get data, usually inaccessible, about the object *o*, such as a representation of its fields, declared and inherited methods. Moreover *mo* can intercept, analyze and possibly change any message sent to *o*. E.g., it can choose wether to pass the message to *o* or not, thus altering the program flow.

Using such facilities in an object-oriented programming language allows the programmer to flexibly manipulate the objects of a running program. These features are very useful to write generic code untied to choices made at compile time, instead allowing the programmer to use information only known at runtime. A typical example is to discover at runtime the list of public constructors available for the class of a given object. The known tradeoff for such capabilities is a possible loss of performance [FF05]. These introspection tools play a central role in the rewriting of the Design Patterns described in chapter 3.

2.1.1 Summary of used reflective Java methods

The object-oriented programming language used in this thesis is Java [AG96]. The purpose of this section is to summarise a part of the reflective API provided by the language.

Java provides a limited, although useful, reflective interface [FF05], in particular it allows[1] the introspection of classes. The language also provides two powerful kinds of support: (i) the capability of loading at runtime classes unknown at compile time, and (ii) the capability of dynamic invocation, i.e. to invoke at runtime a method unknown at compile time.

Some of the classes involved for the reflection operations are `Class`, `Method` and

[1]Without recurring to extensions such as Javassist [Chi00] or Kava [WS00].

`Field`[2], which are made available as standard Java classes, representing (some of) the internal structure held by the Java Virtual Machine (JVM), and allowing their introspection at runtime.

The full API is deeply documented in [Sun07], the following is a brief description of the most important methods that will be used throughout the next technical chapters.

- `Class Class.forName(String)`
  is a static method which returns a `Class` object after, possibly, the load and initialisation of the class into the JVM. For example

  ```
  Class c = Class.forName("java.lang.String");
  ```

  will create an object `c` which represents the `String` class. Please note that `c` is not an instance of `String`.

- `Class getClass()`
  is a method of `Object` (any Java object will inherit it) which returns a `Class` object representing the object on which it is called. For example

  ```
  String s = "test";
  Class c = s.getClass();
  ```

  will retrieve and insert into `c` an object representing the `String` class, as in the previous example, but using object `s` to get the reference.

- `newInstance(...)`
  allows the instantiation at runtime of a class represented by a `Class` object. For example

  ```
  Class c = Class.forName("java.lang.String");
  String s = c.newInstance();
  ```

  will create a new (empty) string in `s`.

- `Method getMethod(...)`
  returns an instance of `Method` representing the method name passed as an argument, as in the next example.

[2] The related package for the classes is `java.lang.reflect`, except for `java.lang.Class`.

- `Object invoke(Object obj, Object[] args)`
  the `invoke()` method belongs to the `Method` class and allows the dynamic invocation of a method unknown at compile time, represented by a `Method` object. Its parameters specify on which object (`obj`) the method has to be invoked and its potential arguments. E.g.

```
Class c = Class.forName("String");
String s = c.newInstance();
Method m = c.getMethod("isEmpty", null);
boolean res = m.invoke(s, null);
```

  will invoke `isEmpty()` on the dynamically created `String` held in `s`.

2.2 Aspect-oriented programming

Object orientation imposes the decomposition of a software system in small, self-contained reusable units (i.e. classes) implementing a (part of a) specific functionality or concern. Instances of these units (i.e. objects) interact with each other exchanging messages (i.e. calling methods).

Unfortunately it is not always possible to partition a software system into purely self-contained classes without spreading the code implementing a concern into more than one class. Such functionalities are called *crosscutting concerns* (*ccc*), as their implementation spreads across the system's classes. Typical examples are the logging, synchronisation and authorisation concerns.

A software system implementing *ccc* usually suffers from *tangling* and *scattering*. The code implementing a *ccc* is written in various classes (it is *scattered*), each of which hosts both its own code (its main, domain, responsibility) and parts of the *ccc* one (*tangling*). This poses many problems for a software system in terms of software engineering, especially for its modularity. Some of the problems that arise are: tightly coupling between classes; code duplication; less reusable code; code more difficult (and expensive) to maintain and understand.

Aspect Oriented Programming [KLM+97] can be seen as a superset of object-oriented programming, adding some conceptual tools to manage the *ccc*s of a software system, allowing an improved modularity. The AOP implementation used in this dissertation is AspectJ [Lad09, hp11], arguably the most mature and developed one, on which both the following description and the whole thesis is based.

To allow the implementation of a *ccc* into a single module the concept of *aspect* is introduced. A system is partitioned in classes (holding the usual business logic) and aspects (holding the *ccc*s). These entities will be recomposed by a specialised bytecode compiler (`ajc`, a *weaver*) to create the final, intended, system. This phase is called *weaving*, and is guided by the rules stated in the aspects.

An aspect can perform both *static* and *dynamic crosscutting*. *Static crosscutting* allows an aspect to modify the structure of existing classes (e.g. adding variable members and methods), so as to allow the programmer to write the code of a *ccc* in a single aspect instead of scattering it throughout the involved classes of the system. *Dynamic crosscutting* deals with the behaviour of the system, allowing the programmer to alter the flow of the program by defining a set of points in the program flow (*pointcuts*) upon which additional code (*advices*) should be executed.

AspectJ offers a set of specialised keywords to implement both kinds of crosscutting. For the sake of understanding this work it is useful to briefly introduce the most important (and used) ones in this dissertation.

An aspect is a module similar to a regular class, which however is automatically instantiated by the system (the programmer can not explicitly instantiate an aspect), which defines *pointcut* and *advices* that implement a *ccc* to be added to the system.

A *pointcut* indicates a set of *join points*. A join point is a well-defined moment in the flow of a program, such as the setting of a member variable, the execution of a method with a certain signature, etc. The weaver is capable of analysing a source program and intercepting the join points defined in a pointcut. A pointcut usually triggers the execution of a related advice, thus altering the regular flow of the program. Pointcuts can be combined using the logical operators `&&` (*and*), || (*or*), `!` (*not*).

An *advice* is a fragment of code (very similar to a regular method), related to some pointcuts, that will be executed when the pointcuts are activated. Given an advice it is possible to let its code execute after, before or instead of the event represented by the pointcut by using, respectively, the `before`, `after()` and `around()` constructs. The additional `proceed()` construct, used only inside `around()` advices, returns the control flow to the captured join point.

A very simple, yet complete, aspect is shown in figure 2.2. The `Hello` aspect logs

```
1  public aspect Hello{
2  
3  pointcut hello():
4        call(void ∗.sayHello (..));
5  
6  void around(): hello(){
7     long start, end;
8     start = System.currentTimeMillis();
9     System.out.println("log: "+start+", before call to sayHello() method");
10    Object res = proceed();
11    end = System.currentTimeMillis();
12    System.out.println("log: "+end+", sayHello() method executed in "+(end−start)+" ms");
13    }
14 }
```

Figure 2.2: *A sample aspect*

on the standard output console the time of any call to any `sayHello()` methods, and keeps track of its running time. The pointcut `hello` takes no parameters and intercepts all the calls to any `sayHello()` method which returns nothing (i.e. `void`) and that has zero or more parameters. Lines 6–13 show the related advice definition. This code will be bind by the weaver to the application classes, i.e. the code will be inserted into the join points satisfying the `hello` pointcut.

The basic keywords used to capture execution of some methods are `call()` and `execution()`. Some constructs used in the next chapter are presented in the following.

- `call()`
  the pointcut `call(int Account.refresh())` collects at runtime all the calls of the `Account.refresh()` method returning an `int`. The context of the call is in the scope of the caller object.

- `execution()`
  the pointcut `execution(int Account.refresh())` collects at runtime the execution of the `refresh()` method. This is different from the previous one as the collected context is in the scope of the `Account` object executing the method.

- `set()` and `get()`
  respectively intercept the reading or writing of a member variable.

- `within()`
  collects all the join points happening inside a scope: `within(Account)` captures all the join points inside the `Account` class.

Additional constructs are made available by AOP languages to allow the code of the advices to be able to access the context on which they are injected.

- `this()` – provides a reference to the object executing the captured join point.
- `target()` – provides a reference to the object receiving a method call.
- `args()` – provides a reference to the values of parameters of the captured invocation.
- `@target(Deprecated)` – filters out all method calls whose target is annotated with the `@Deprecated` annotation.
- `thisJoinPoint` – provides various forms of reflective access to the dynamic context of the captured join point, e.g. the signature of the captured join point.

2.3 Metadata and annotations

Many programming languages offer support for managing metadata, i.e. additional data about the program itself that usually do not affect its execution. One of the simplest metadata supported arguably by any programming language are comments in the source code, usually discarded in the compilation phase. The Java language supports metadata by means of annotations, a structured way integrated with the language to deal with metadata.

Annotations are written in the code usually to mark classes, methods and fields with specialised meaning. Annotations provided by the Java language allow, e.g., to disable warnings at compile time or to force the override of a method.

Annotations can have parameters and can be user-defined. Figure 2.3 shows such an annotation, which defines an `@Proxy` annotation with a `String` parameter named `value`. A class can be annotated as follows:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Proxy {
    String value();
}
```

Figure 2.3: *User-defined annotation to mark a Proxy role*

```
@Proxy("Checker") public class Account { ... }
```

Such an annotation marks the `Account` class as part of a *Proxy* design pattern. In particular it defines the `Checker` class as a Proxy for the `Account` class[3].

[3]The detailed explanation can be found in section 3.3.

Chapter 3

Aspect-oriented design patterns

This chapter deals with the aspect-based design pattern implementations proposed, presenting them in different versions. All the basic object-oriented patterns discussed are originally described in [GHJV94].

The main contribution is the *Aspect-Oriented and Annotated* (AA) version of a design pattern. A design pattern written in the AA version takes full advantage of the tools described in chapter 2, i.e. AOP, computational reflection and annotations. A design pattern is encapsulated in a general aspect to be woven into *any* application needing some classes to implement the design pattern's roles. The full generality of the aspect is obtained by means of reflection, gaining any additional information needed to apply the advices at runtime. A class playing a role for a design pattern is usually marked with a provided user-defined annotation specifying its role and possibly additional parameters. The triggered advices of the aspect implement the behaviour of the specific roles of the pattern.

The generality of an AA aspect allows the complete reusability of the aspects in any context, as the aspects do not mention any specific class names in their code, any needed information is gathered at runtime by means of reflection. However, the use of reflection might become a limit for the system performance [FF05], so the *Cached, Aspect-Oriented and Annotated* (CA) version is proposed as an improved, functionally equivalent, alternative over the AA implementation. A thorough assessment of the performance of all the variants of the approach is presented in chapter 5.

```
public @interface Singleton { }
```

Figure 3.1: *Annotation for the* Singleton *design pattern*

3.1 *Singleton*

The *Singleton* design pattern describes a solution for limiting the number of instances of a class, tipically allowing just one instance.

In the regular object-oriented solution, to render an existing `C` class a Singleton, `C` has to be modified to expose no public constructors, the only access point for the class is a specifically written static method, usually `getInstance()`, that implements the instantiation logic. All the clients need to pass through this method to get an instance of the Singleton class. The constructors made private are still accessible by the `getInstance()` method, that is allowed to create a new instance of the class or return the existing one to the caller (client).

The changes in the `C` class are reflected in all the clients' classes, as they are tightly coupled with the Singleton class, i.e. a client class must be aware of the role played by `C` to be able to use it. Thus, any class accessing `C` must use the `getInstance()` method instead of the regular constructor. So when a Singleton class will not play this role in future evolutions of the software, all clients accessing it must also be changed to use the regular constructor (instead of the `getInstance()` method).

When a class is (or becomes) a Singleton a new responsibility is superimposed on its main concern. The `C` class will not just implement its main domain code but also the Singleton-related code for the management of its unique instance. This additional code makes the class less reusable and more complex to understand. Moreover the code of the `getInstance()` method that must be implemented is essentially the same for any class that has to behave as a Singleton.

3.1.1 Aspect-oriented and annotated *Singleton*

The `@Singleton` annotation (figure 3.1) is a tagging annotation, as it bears no parameters, and is used to impose the Singleton behaviour on any `C` class without any changes to its constructors nor requiring the `getInstance()` method.

For instance, to make a `Bank` class a Singleton it is sufficient to annotate it as follows

```
@Singleton public class Bank { ... }
```

leaving the original constructor visibility as it is (i.e. there is no need to make it private) and also without explicitly writing a `getInstance()` method. The provided aspect will implement the (apparently) missing Singleton behaviour (satisfying *ECoR*).

The `SingletonPattern` aspect (figure 3.2) intercepts any `new` invocation on any `C` class marked with the `@Singleton` annotation, checks whether the call for a new `C` object is the first one or not, and, respectively returns a new `C` object or the already instantiated (unique) instance of `C`. No changes need to be made on clients' classes.

When, e.g., the annotated class is the `Bank` one, clients will access the `Bank` class by using its public constructor, i.e. making a `new` invocation, however this instruction will be intercepted by the aspect that will make sure that the actual instantiation happens just once. Any client can transparently access the Singleton class without knowing whether it is playing a Singleton role or not, thus making clients independent of the Singleton role.

For example, using the provided aspect, a client instead of invoking a static method, as in the following

```
Bank b = Bank.getInstance();
```

will just invoke

```
Bank b = new Bank();
```

as the aspect will allow or disallow such an invocation taking care of the number of instances of the `Bank` class.

This approach is also robust enough, thanks to the nature of the aspect-oriented technology. The programmer can not make the mistake of creating more than one instance of the `Bank` class, as the uniqueness of the instance will be guaranteed by the `SingletonPattern` aspect, in fact for successive `new` invocations the aspect will provide the same reference. This satisfies *REoR*.

Thus, in general, a Singleton class can be used as such without burdening the programmer to explicitly write code for such a role. To allow a class to behave as a Singleton all the programmer has to do is to weave the `SingletonPattern` aspect

```
public aspect SingletonPattern {
    private Hashtable<Class, Object> singles = new Hashtable<Class, Object>();

    pointcut trapCreation(): call((@Singleton *).new(..));

    Object around() : trapCreation() {
        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 3.2: *The* SingletonPattern *aspect*

singles	
`Class`	`Object`
Bank	Bank@739
Spooler	Spooler@255

Table 3.1: *Sample values for the* singles *map*

with its own application code (thus satisfying *SPoC*), by appropriately annotating the class. The `Bank` class will have its regular constructor and will be reusable, without changes, in other contexts which might not require it to play a Singleton role (satisfying *SoC*).

3.1.2 `SingletonPattern` aspect

The `SingletonPattern` shown in figure 3.2 keeps the `singles` map which stores the references to all the Singleton classes' instances, indexed by class. As a desired characteristic of the aspect is its generality, the values stored into the map are of the `Object` class, so to accommodate any possible object type to play the Singleton role.

The aspect is composed of just one pointcut, `trapCreation`, and its related

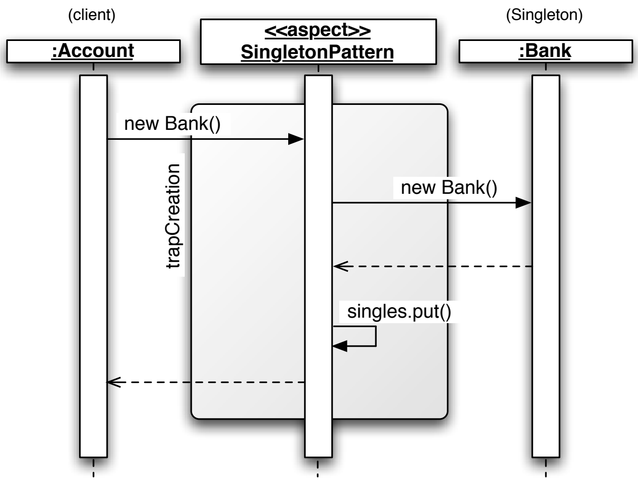

Figure 3.3: *A* Singleton *created, and stored, by the aspect*

advice. The `trapCreation` pointcut (line 6) intercepts any constructor (i.e. `new`) invocation on any class marked with the `@Singleton` annotation.

When the `trapCreation` advice is triggered, it interrupts the captured `new` call and stores its target class in `s`. This is done using reflection (line 8), obtaining the class using the `getDeclaringType()` method on the signature of the intercepted join point.

If a `Bank` class is annotated as a Singleton, a call from any client yields `thisJoinPoint` to become `call(Bank())`, so `getDeclaringType()` returns a reference to the `Bank` class, i.e. the class on which the intercepted `Bank()` constructor is declared.

Next, it will be checked if an instance of the (dynamically retrieved) `Bank` class has already been created (line 9), by looking for a `Bank` key in the `singles` map; an example of a populated `singles` map is shown in table 3.1. Two possibile scenarios are: there is no instance of the Bank class (figure 3.3) or an instance has already been created (figure 3.4).

In figure 3.3 an `Account` object tries to instantiate a `Bank` object marked as Singleton. The `new` invocation is intercepted by the `SingletonPattern` aspect and the `if` instruction on line 9 evaluates to `true`, so the `proceed` instruction executes the intercepted `new` call, passing the message to the `Bank` class. The newly created `Bank` object is stored in the `singles` map, so to return it as the unique instance for any future call, as in the next case. In the end the created object is returned to the client (line 14).

Supposing that the same `Account` client tries to instantiate another `Bank` object,

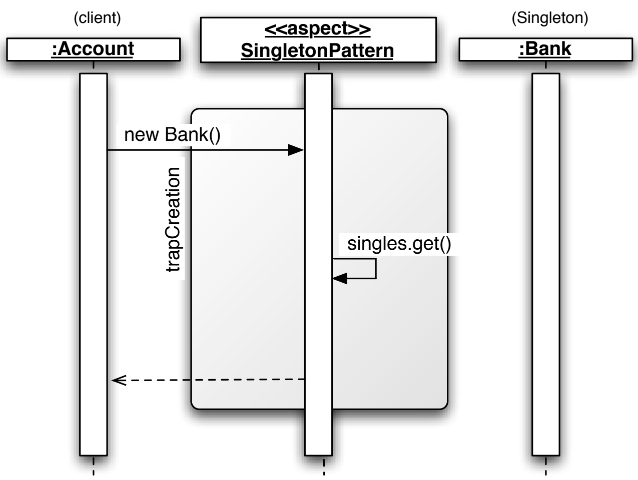

Figure 3.4: *A* Singleton *is not created but retrieved from the aspect*

the behaviour of the `SingletonPattern` for this scenario is depicted in figure 3.4. This time control in line 9 yields `false`, later (line 13) the `obj` variable will contain the reference for the `Bank` object already created, this reference is returned to the invoking client, so no call to the `Bank` constructor is made and the caller gets the only instance of the `Bank` class already instantiated.

The `SingletonPattern` aspect makes use of two reflective calls in line 8; such calls are repeated for any intercepted `new` call to a Singleton class. So the aspect could be rewritten in order to allow the caching of the results of these repeated reflective calls. This rewriting has been done for other design patterns (e.g. see *Proxy* in 3.3.3), however no practical benefits in terms of execution time would be achieved for this very AODP.

For the implementation of such a cache, the `getSignature()` call can not be avoided, as it is needed to tell apart what join point has been intercepted, but the result of `getDeclaringType()` could be cached in a map, say `sigs`, indexed by the `Signature` obtained from the previous call. However, to retrieve the `Class` from a `Signature` it is necessary an access to the `sigs` map instead of the call to the reflective method. Unfortunately, this exchange of calls gives no performance gain, as preliminary tests have shown that the cache version, for this pattern, is between 9–12% slower than the AA version, thus it has not been used.

A variant of the described `SingletonPattern` is the *Limiton* variant [SW08] of the *Singleton* pattern, where the number of instances of the class acting as a Singleton is limited to n. The annotation used for this version is shown in figure 3.5.

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Singleton {
    Integer n();
}
```

Figure 3.5: *Annotation for the* Limiton *variant*

The `n` parameter on an annotated `C` class is used by the aspect as an upper bound on the number of instances of `C`.

3.2 *Flyweight*

The *Flyweight* design pattern is mainly used in a performance-aware environment, as it provides a solution to the instantiation of many objects of the same class, say `C`, possibly exhausting the system memory.

The solution suggests to partition the members of the `C` class in two sets: a set representing the *intrinsic state* of `C` and a set for the *extrinsic state*. The former is a set of attributes of proper data about C, independent of the context on which an object will be used, such as a character, while the latter includes any other information about the context where the former will be used, e.g. the position or the font size for the character. The intrinsic state is stored in a new class `C'`, playing the ConcreteFlyweight role, the extrinsic state is stored in another class, say `E`. This partition allows clients to share the same object for the part of the state of an object that remains the same in any context where it can be used.

Any client class can not directly instantiate a ConcreteFlyweight object, instead it has to ask a FlyweightFactory for a reference. The FlyweightFactory, given a key to identify the requested object, returns the reference to the object corresponding to the passed key. The returned object conforms to the *Flyweight* interface, implemented by the ConcreteFlyweights. It is a responsibility of the client class to provide the received ConcreteFlyweight with the necessary extrinsic state so to be able to properly use it.

Using this design pattern, it is self-evident how any client class have to be tightly coupled with the FlyweightFactory class to be able to instantiate any ConcreteFlyweight object, thus hindering its reusability when the ConcreteFlyweight class should not play that role anymore for evolution purposes.

```
public @interface Flyweight { }
```

Figure 3.6: *Annotation for the* Flyweight *design pattern*

```
public aspect FlyweightPattern {
    private Map<Class, Map<Integer, Object>> flyws =
                        new Hashtable<Class, Map<Integer, Object>>();

    pointcut trapCreation(Object a): call((@Flyweight *).new(..)) && args(a);

    Object around(Object a): trapCreation(a) {
        Integer hash = MyHashing.getHash(a);
        Class targetFlyw = thisJoinPoint.getSignature().getDeclaringType();
        Map<Integer, Object> keys = flyws.get(targetFlyw);
        if (keys == null) keys = new Hashtable<Integer, Object>();
        if (keys.containsKey(hash)) return keys.get(hash);
        Object ref = proceed(a);
        keys.put(hash, ref);
        if (keys.size() == 1) flyws.put(targetFlyw, keys);
        return ref;
    }
}
```

Figure 3.7: *The* FlyweightPattern *aspect*

Moreover the FlyweightFactory has to be specifically written every time a new *Flyweight* design pattern has to be implemented. Its code is basically the same as its responsibility is to check if, given a key, the corresponding instance has already been created or not (and then return its reference to the client), however it depends on the class playing as ConcreteFlyweight.

3.2.1 Aspect-oriented and annotated *Flyweight*

The implementation of the *Flyweight* design pattern detailed in the next section uses the `@Flyweight` annotation (figure 3.6) to mark any class that should play the ConcreteFlyweight role. When an application is woven with the `FlyweightPattern` aspect (figure 3.7) the latter will intercept any `new` call to any class annotated with the `@Flyweight` annotation. The management of the ConcreteFlyweights is done by the `FlyweightPattern`, thus satisfying the *ECoR* property, which can be disabled

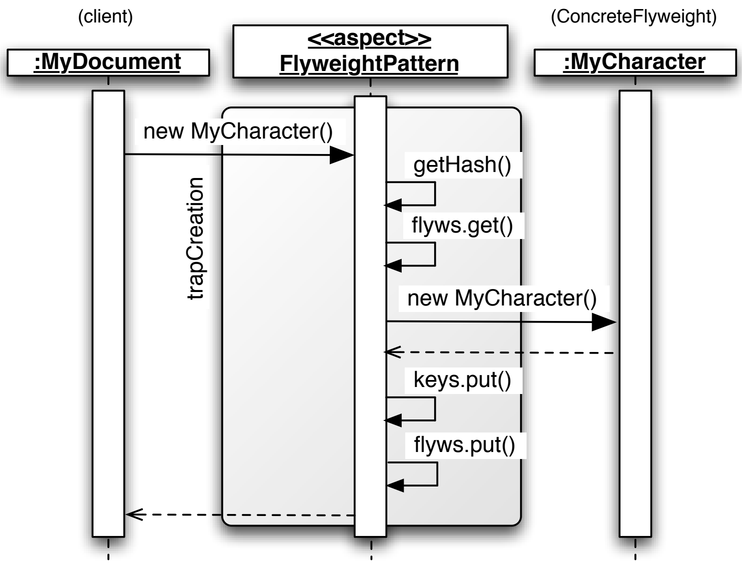

Figure 3.8: *A* ConcreteFlyweight *is created, and stored, by the aspect*

just by removing the aspect from the application (fulfilling the *SPoC* property).

A client does not need to resort to any FlyweightFactory to receive an instance of a ConcreteFlyweight, given that the ConcreteFlyweight class has been properly annotated. For example, to make the `MyCharacter` class a ConcreteFlyweight, the class has to be annotated as follows

```
@Flyweight public class MyCharacter { ... }
```

so that any client instead of invoking

```
MyCharacter c = MyCharacterFactory.getMyCharacter();
```

can just invoke

```
MyCharacter c = new MyCharacter();
```

thus making it not coupled anymore with the `MyCharacterFactory` class.

Using the `FlyweightPattern` aspect the programmer still needs to manually separate the intrinsic and extrinsic state from the original class to be made a ConcreteFlyweight, as such a "semantic" partitioning can not be automatically performed. However, the FlyweightFactory behaviour is transparently provided by the `FlyweightPattern`, which bears the responsibility to provide the shared instances to any client invoking `new`, thus fulfilling *REoR*. This allows the client classes to remain unchanged when the `MyCharacter` class does not play the ConcreteFlyweight role anymore.

All the code for instance management is held in the `FlyweightPattern` aspect, which, thanks to the use of reflective constructs, is completely reusable and allows the class acting as ConcreteFlyweight to implement just its domain code (thus fulfilling *SoC*).

3.2.2 `FlyweightPattern` aspect

The code for the `FlyweightPattern` is shown in figure 3.7. The aspect uses the `flyws` map to store the references to the ConcreteFlyweight already instantiated. To identify an instance of a ConcreteFlyweight class, a pair (`Class`, `Integer`) is used. The class of the requested ConcreteFlyweight is used paired with an integer number computed as a hash of the parameters used by the client to create the ConcreteFlyweight.

As the aspect has to be generic, the map has to accommodate any possible class instance, thus the value for the map is declared as an `Object` type. The map is declared as a `Hashtable` (instead of a `WeakHashMap`), as the references it stores have to be kept even if there is no object of the application having any reference, i.e., as a policy, when an object is created it remains in memory, ready to be returned to any client.

The `trapCreation` pointcut intercepts any `new` call to any class marked with the `@Flyweight` annotation, in the same way as it happens in the `SingletonPattern`. The similarity they share comes from the fact that both have to manage a limited number of instances of classes. In this case however there is more than one possible instance per class, as, at most, for each ConcreteFlyweight class there will be an instance for every possible value of the parameters (for the constructor) of the class. Figure 3.8 depicts a possible interception of a ConcreteFlyweight creation captured by the aspect.

The first operation performed in the advice (line 8) is the computation of an hash for the parameters passed to the intercepted `new` call by the client, to identify which object of the ConcreteFlyweight class has been requested.

The computation of the hash for the parameters is done by the `MyHashing.-getHash()` method. If the input parameter is a primitive type its hash is computed by the `hashCode()` method inherited by any Java object. For any other parameter type a map (`pars`) with all the types and values of its variable members is constructed and its resulting hash is the `hashCode()` of the `pars` map. Please note that [GHJV94] does not impose, nor show, a general way to deal to the identification of the ConcreteFlyweights, the one proposed here is just a possible general way to solve this problem[1].

In line 9 the target class of the intercepted `new` is computed in the same way as in the `SingletonPattern`. The remaining lines checks if an object for the (class, hash) pair has already been computed or if it has to be created (and thus let the intercepted pointcut to be executed by the `proceed()` statement in line 13) and inserted in the `flyw` map before returning its reference to the client.

The `FlyweightPattern` aspect bears some resemblance to the `SingletonPattern` one in the usage of reflective calls (cf. figure 3.7 line 9 with figure 3.2 line 8). Thus the use of a map to cache repeated calls brought similar conclusions about the caching of the `getDeclaringType()` method results. As in the `SingletonPattern` case a caching mechanism offers no benefits in terms of speeding up the execution time of the design pattern code, in this case preliminary tests have shown that the cache version of the `FlyweightPattern` aspect becomes about 5% slower than the regular one.

3.3 *Proxy*

The *Proxy* design pattern describes a way to shield an object from direct access from other clients' objects, as they access just a substitute object for any message to be sent to the shielded one.

The shielded object plays the RealSubject role, the substitute object plays the Proxy role. All the accesses to the RealSubject pass through the Proxy, which exposes the same interface of the RealSubject and it is the only entity allowed to access it. Both Proxy and RealSubject implement the same Subject interface, so that clients can use the same methods to access the Proxy as they would to access the RealSubject.

[1] In [GHJV94] a generic `key` parameter is used to identify a ConcreteFlyweight instance. If such a key is just a `char`, as in the [GHJV94] sample code, it can directly be used as an index for a map of ConcreteFlyweights, however a key could be any arbitrary object, so it has to be treated accordingly, as in the proposed solution.

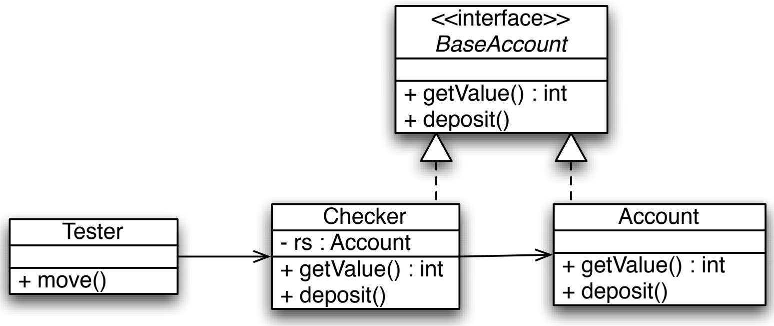

Figure 3.9: *Sample application using an object-oriented* Proxy

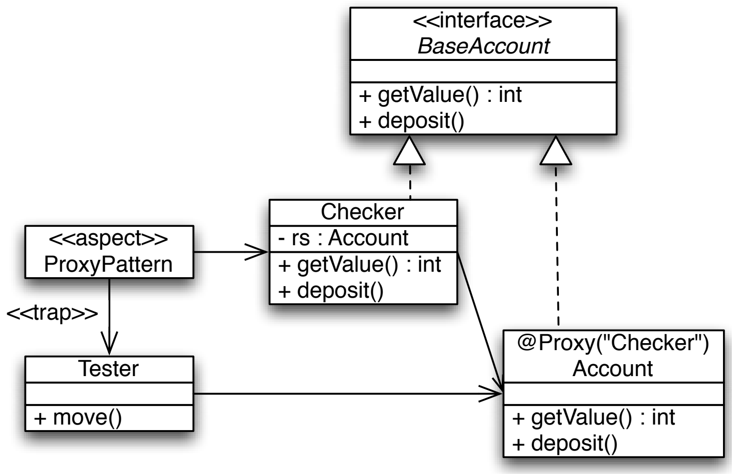

Figure 3.10: *Sample application using an AA* Proxy

It is a responsibility of the Proxy to forward any message to its RealSubject. For instance, a Proxy might cache the results of a computation and return these cached results instead of making the RealSubject redo the computation.

Figure 3.9 shows an UML class diagram for a sample application implementing a *Proxy*. A client class `Tester` accesses the `Checker` class (Proxy) instead of `Account` (RealSubject); both RealSubject and Proxy implements the same interface `BaseAccount`.

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Proxy {
    String value();
}
```

Figure 3.11: *Annotation for the* Proxy *design pattern*

3.3.1 Aspect-oriented and annotated *Proxy*

For the AA version of the *Proxy* design pattern a special annotation `@Proxy` is defined, shown in figure 3.11, to be used together with the aspect shown in figure 3.12. With this facility, for any application class `C` to play role RealSubject behind a Proxy class `C'`, it is sufficient to have `C` marked with the annotation `@Proxy(C')`. Once aspect `ProxyPattern` has been woven into the annotated class `C`, it will enforce the RealSubject behaviour expected of `C`. This fulfils property *SoC*. Moreover, client classes need not be changed to let them invoke methods of Proxy `C'` (instead of `C`), satisfying *SPoC*.

A simple example is the object-oriented application in figure 3.9. To make use of the AA version of *Proxy*, to inhibit clients direct access to `Account`, thus making it play as a RealSubject, the class just needs to be annotated as follows:

```
@Proxy("Checker") public class Account { ... }
```

The resulting UML-like notation for the AA version is shown in figure 3.10. Unlike the object-oriented version of *Proxy*, in the AA version of *Proxy*, client classes need not explicitly invoke methods on instances of the class playing Proxy, but keep referring to the instances of the class playing RealSubject. Then, it is the intervening aspect that shields RealSubject and forces the use of Proxy. E.g., to access class `Account` playing as a RealSubject, a client class `Tester` would use a code fragment like:

```
Account acc = new Account();
acc.getBalance();
```

Upon the execution of the above `new Account()`, the general aspect `ProxyPattern` will: (i) intercept instantiation of `Account` (the RealSubject); (ii) create an instance of the associated Proxy, i.e. `Checker`; and finally (iii) pair the two just created instances. Then, whenever method `getBalance()` is called on the created instance `acc` of RealSubject `Account`, the said aspect will intervene and the namesake method on the corresponding instance of Proxy `Checker` will be invoked instead.

Through this approach, client classes are unaware of Proxy and the type of the variable holding a RealSubject (i.e. `acc` of the example) remains unchanged even though Proxy is used instead. This accommodates property *REoR*.

Moreover, if the choice to shield an application class behind a Proxy is abandoned for evolution purposes, the `@Proxy` annotation is simply removed from it, and remaining application code is unaffected, thus ensuring *SPoC*. This also makes client classes reusable in different contexts, as their code is not coupled with the Proxy class. E.g., the caller code in the previous fragment is not affected if the `Checker`–`Account` pair is separated. Thus, property *ECoR* is fulfilled.

3.3.2 `ProxyPattern` aspect

To implement the desired behaviour, the `ProxyPattern` in figure 3.12 defines two pointcuts, `trapCalls` and `trapCreation`, each with an associated advice. The advices respectively handle any method call or any constructor call to any RealSubject.

The `proxies` map[2] stores the objects' pairs (RealSubject, Proxy), as the aspect needs to link each RealSubject instance with its (automatically created) Proxy. Please note that the `proxies` map is defined to hold just `Object` references, thus it is capable of holding any class for both RealSubject (the key of the map) and Proxy (the value of the map).

The pointcut `trapCalls` intercepts any method call on objects whose class is marked with the `Proxy` annotation, so effectively intercepting any call to any RealSubject object, provided that its class has been properly annotated. It also collects references about the context of the call: the caller object `t` (by using `this`), the invoked object `o` (by using `target`) and `a` as the annotation on the callee (through `@target`).

The responsibility of the `trapCalls`-related advice is to redirect a call on a RealSubject to a Proxy, as explained before. The advice behaviour depends on the context of the interception, in particular on the type of the caller `t` (see the if statement in line 9 of figure 3.12). If `t` is a RealSubject calling its own method,

[2]Please note that `proxies` is initialised as a `WeakHashMap` so to allow the garbage collector to reclaim unused memory should a RealSubject become a null reference.

```
public aspect ProxyPattern {
   private Map<Object, Object> proxies = new WeakHashMap<Object, Object>();
   private Object tmp = null;

   pointcut trapCalls(Object o, Object t, Proxy a):
         call(* *.*(..)) && target(o) && this(t) && @target(a);

   Object around(Object o, Object t, Proxy a) : trapCalls(o, t, a) {
      if ((t == o) || (t.getClass().getName().equals(a.value())))
         return proceed(o, t, a);
      try {
         Class c = Class.forName(a.value());
         MethodSignature s = (MethodSignature) thisJoinPoint.getSignature();
         Method m = c.getMethod(s.getName(), s.getMethod().getParameterTypes());
         return m.invoke(proxies.get(o), thisJoinPoint.getArgs());
      } catch (Exception e) { /* ... */ }
      return null;
   }

   pointcut trapCreation(Object t):
         call((@Proxy *).new(..)) && this(t);

   Object around(Object t): trapCreation(t) {
      Proxy ap = (Proxy) thisJoinPoint.getSignature()
                  .getDeclaringType().getAnnotation(Proxy.class);
      if (t.getClass().getName().equals(ap.value())) return tmp;
      tmp = proceed(t);
      try {
         proxies.put(tmp, Class.forName(ap.value()).newInstance());
      } catch (Exception e) { /* ... */ }
      return tmp;
   }
}
```

Figure 3.12: *The* ProxyPattern *aspect*

the call will proceed without any other effect. The same applies if the caller is the Proxy for the RealSubject o; these are verified by comparing the caller's class name with the class name found as a parameter of the annotation on the RealSubject (i.e. `a.value()`).

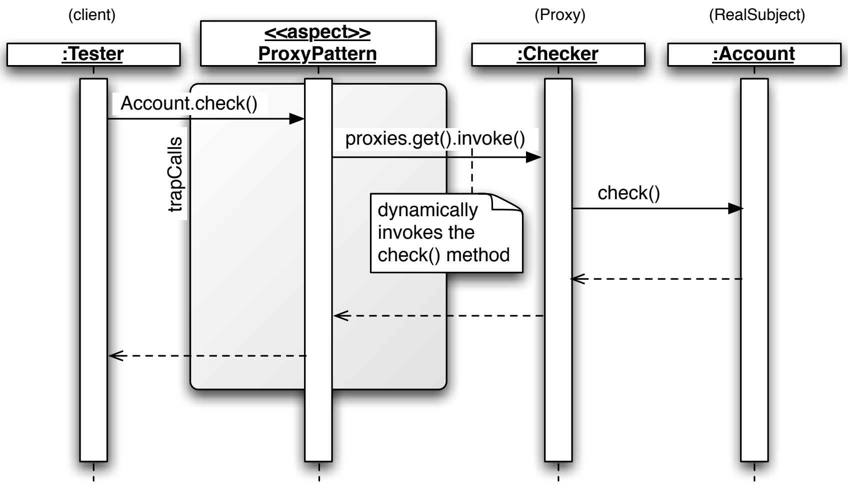

Figure 3.13: *A call dynamically passed to the* RealSubject

Any other call captured by the `trapCalls` pointcut is treated as a regular client call to a RealSubject object, so the method call has to be invoked on the Proxy instead of invoking it on the RealSubject. This scenario, with the Proxy invoking the call on its RealSubject is depicted in figure 3.13.

Since the `ProxyPattern` aspect is generic, it bears no reference to any specific method to invoke when the `trapCalls` pointcut activates. The method name to invoke on the Proxy is retrieved at runtime by introspection. Specifically, the name and parameters of the captured method, i.e. invoked by the caller `t` (a client), are discovered and then used to dynamically invoke the namesake method on the Proxy (see lines 12–15, figure 3.12). The name of the captured method is discovered using `getSignature()` on `thisJoinPoint`, for the dynamic invocation a `Method` object is created, using the name and parameters of the captured one, then it is invoked on the Proxy paired with the RealSubject `o`. The correct Proxy on which to invoke the target method is retrieved from the `proxies` map, which is populated as explained later.

Please note that the scenario depicted in figure 3.13 is not the only one possible, as the call from a Proxy to its RealSubject depends on the logic of the method implemented within the Proxy, e.g. if the Proxy acts as a caching device, it might not forward the invocation to its RealSubject but would just return the cached return value.

Populating the `proxies` map

The advice corresponding to pointcut `trapCalls` works as described as long as the `proxies` map is correctly initialised, the responsibility for such initialisation is described in the following and depicted in figure 3.14.

The `trapCreation` pointcut intercepts any `new` invocation on a class bearing the `@Proxy` annotation. The only context to be collected in this case is the caller object `t`, the client requesting a new RealSubject object. The related advice provides the (transparent for the client) connection between a RealSubject and its Proxy, populating the `proxies` map.

Once activated, the advice obtains, via AspectJ facilities and reflection, the parameter of the `@Proxy` annotation found in the RealSubject class whose instatiation has been intercepted. This is the Proxy's class name, stored in the `ap` local variable (line 24, figure 3.12). When the caller is a regular client (i.e. not a Proxy), unaware of the Proxy intervention, the instantiation is allowed (`proceed` on line 27 in figure 3.12) and locally stored in `tmp`. Using reflection a new Proxy is created and linked to the just created RealSubject, putting the pair in the `proxies` map. The Proxy class to be (possibly) loaded at runtime, is obtained using `Class.forName()`, and is identified by the value of the `@Proxy` annotation retrieved in `ap`. A new instance is dynamically created using `newInstance()`. Control is now returned to the client, with the `proxies` map now ready to respond to future calls of methods on the newly created RealSubject by routing them to its unique Proxy.

Advice execution can also be triggered when `new` on a RealSubject is executed from a Proxy. This case is similar to the `trapCalls`' advice, i.e. the Proxy is allowed to instantiate a RealSubject. By the very nature of a Proxy, the instantiation of its RealSubject might be deferred in time or might not even happen, this depends on the code of the Proxy. Although the RealSubject has already been created by the provided aspect, nothing bad happens if that would be the case, as the already created instance will be returned to the Proxy, when at later time the Proxy executes `new`. The `trapCreation` advice is triggered again and the `if` (line 26) evaluates to `true`, thus since `tmp` still points to the RealSubject instance to be associated with the invoking Proxy instance, the latter is simply returned `tmp`.

Please, once again, note how the behaviour of the `ProxyPattern` aspect render it completely general and unaware of the environment in which it will be woven

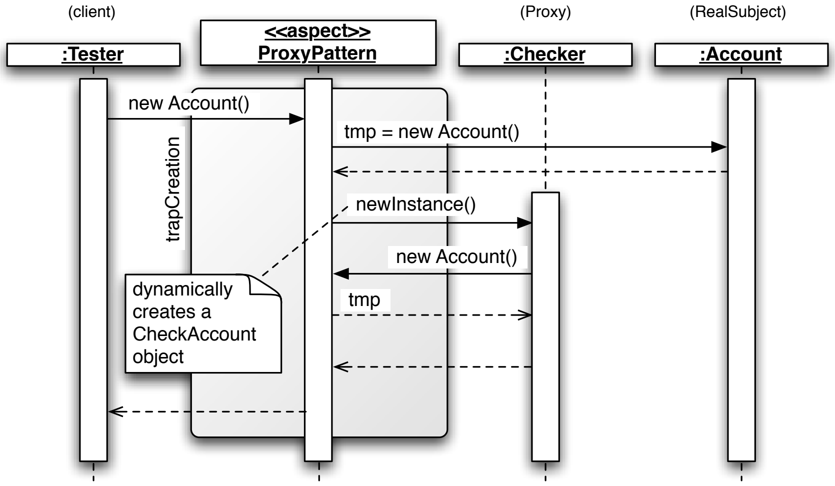

Figure 3.14: *Automatic instantiation of a* Proxy

into. Any information needed to link the aspect's behaviour to any application is reflectively acquired at runtime, while the weaving is driven by the annotations that mark the RealSubject classes.

3.3.3 `ProxyPatternCA` aspect

As shown in the previous section, the AA `ProxyPattern` makes an extensive use of reflective calls to be as general as possible. However, its generality might become a burden to the performance of the application using it, so an enhanced (and refactored) version is provided, i.e. the cached version, shown in figures 3.15 and 3.16.

The CA version retains the generality, the behaviour and the activation logic (i.e. the pointcuts) of the AA version while allowing the code to run faster. The following depicts just the differences with the AA version.

The `ProxyPatternCA` aspect uses three additional maps (`classes`, `mets` and `rss`) and a list (`rs`); the latter is an enhanced replacement for the `tmp` variable in figure 3.12.

The `classes` map is used to limit the number of invocations to the reflective method `getClass()` (line 9, figure 3.12). In the AA version, the `trapCalls`-related advice has to call the `getClass()` method in order to go on, so if the same client (as Proxy or as a regular client) executes the same method several times, the advice will need to call `getClass()` each time, to compute the client's class. However, the result of `getClass()` is the same when applied to the same object, so, given the

```
public aspect ProxyPatternCA {

  private Map<Object, Object> proxies = new WeakHashMap<Object, Object>();
  private Map<SourceLocation, Method> mets = new Hashtable<SourceLocation, Method>();
  private Map<Object, String> classes = new Hashtable<Object, String>();
  private Map<Object, Object> rss = new Hashtable<Object, Object>();
  private LinkedList<Object> rs = new LinkedList<Object>();

  pointcut trapCalls(Object o, Object t, Proxy a):
      call(* *.*(..)) && target(o) && this(t) && @target(a);

  Object around(Object o, Object t, Proxy a): trapCalls(o, t, a) {
      if ((t == o) || (t==proxies.get(o)) ) return proceed(o, t, a);
      return invokeOnProxy(thisJoinPoint, proxies.get(o));
  }

  pointcut trapCreation(Object t):
      call((@Proxy *).new(..)) this(t);

  Object around(Object t): trapCreation(t) {
      String s = ((Proxy) thisJoinPoint.getSignature().getDeclaringType().
                                      getAnnotation(Proxy.class)).value();
      if (isCallerProxy(t, s)) {
              if (rss.containsKey(t))
                  return rss.get(t);
              return rs.peek();
      }
      rs.add( proceed(t));
      try {
          Object p = Class.forName(s).newInstance();
          proxies.put(rs.peek(), p);
          rss.put(p, rs.peek());
      } catch (Exception e) { /* ... */ }
      return rs.poll();
  }
}
```

Figure 3.15: *The* ProxyPatternCA *aspect (part 1 of 2)*

```
private boolean isCallerProxy(Object t, String s) {
    if (proxies.containsValue(t))
        return true;
    String c = t.getClass().getName();
    classes.put(t, c);
    return c.equals(s);
}

private Object invokeOnProxy(JoinPoint jp, Object p) {
    SourceLocation s = jp.getSourceLocation();
    try {
        if (mets.containsKey(s))
            return mets.get(s).invoke(p, jp.getArgs());
        Method m1 = ((MethodSignature)jp.getSignature()).getMethod();
        Method m = p.getClass().getMethod(m1.getName(), m1.getParameterTypes());
        mets.put(s, m);
        return m.invoke(p, jp.getArgs());
    }
    catch (Exception e) {
        /* ... */
        return null;
    }
}
```

Figure 3.16: *The* ProxyPatternCA *aspect (part 2 of 2)*

same client, all invocations except the first one can be avoided. Hence a caching device is used to store the result of the `getClass()` invocation the first time it is executed and later the stored value is retrieved when the same client is identified (method `isCallerProxy()` in figure 3.16).

The `mets` map is used by the `invokeOnProxy()` method as a cache memory for reflective calls, so to allow it to retrieve references to already intercepted methods. The `invokeOnProxy()` method encapsulates the basic behaviour of the `trapCalls`-related advice shown in figure 3.12. It takes two input parameters: `jp` as the reference to the join point intercepted by the `trapCalls` pointcut, and `p` as the reference to the Proxy on which the intercepted method shall be invoked. It encapsulates the main responsibility of the `trapCalls`-related advice, that is to invoke on the Proxy the intercepted method. However, there are two main differences between these implementations.

The biggest difference is the use of the `mets` map to avoid unnecessary repetitions of the computations in lines 14–15 of figure 3.16. These lines respectively compute the method intercepted by the `trapCalls` advice (i.e. the method directly invoked on the RealSubject by the client: `m1`) and the reference to the method of the same name declared on the Proxy `p` (i.e. `m`), in the same fashion as in figure 3.12. However, once such a reference is computed it is also stored in the `mets` map, so to be directly accessible in (possible) future calls. The key used to identify an already executed method is its `SourceLocation`, i.e. the line of code where the method is defined in its class.

Apart from the use of an additional cache memory, the other difference with the original `trapCalls` advice is its main if statement. The original one (line 9, figure 3.12) needs to use reflective code, this is avoided in this enhanced version (line 13, figure 3.15).

The purpose of the conditional statement, as in the AA case, is to tell apart the nature of the caller of an intercepted method on a RealSubject. To check if the caller is a `RealSubject`, and so allowed to invoke its own methods, the same reference comparison will be performed in both AA and CA cases.

To discover whether the caller of the intercepted method is a Proxy is now performed by checking for the existence of the caller reference on the `proxies` map. This is possible because, whenever a new Proxy object is created, it is also inserted into the `proxies` map, so it is sufficient to compare the caller with the (possibly) existing Proxy for the RealSubject on which the captured method was directed to, i.e. the reference obtained using `proxies.get(o)`.

The `rss` map is used in the `trapCalls` advice, as a facility to store the pairs (Proxy, RealSubject) for a Proxy that defers the creation of its own RealSubject, i.e. when the RealSubject is not created inside its own constructor.

The following scenarios detail two possible cases for the creation of the RealSubject by a Proxy, thus thoroughly explaining all the creation logic implemented in the `ProxyPatternCA` aspect. The examples use the class structure already shown in figure 3.10.

The first scenario depicts a Proxy, `Checker`, which creates its own RealSubject, `Account`, inside its own constructor (figure 3.17). This scenario is triggered when a `Tester` (i.e. a client) class invokes `new` on an `Account` (i.e. RealSubject) class.

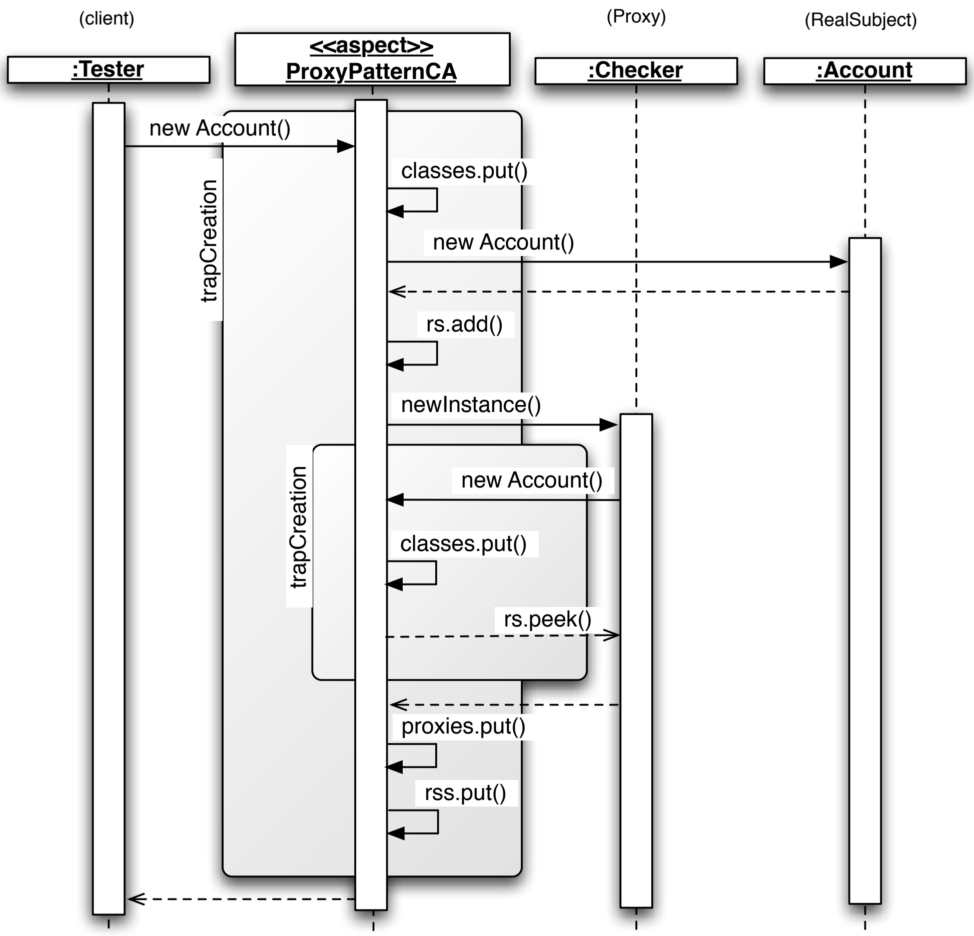

Figure 3.17: *A* Proxy *creating its own* RealSubject *in its own constructor*

The `ProxyPatternCA` aspect is activated and executes the `trapCreation` advice. The `isCallerProxy()` call (line 23) yield `false`, as the `new` call comes from from the `Tester` class. Moreover, its execution populates the `classes` map, e.g., with the pair (`Tester@739`, `"Tester"`). `trapCreation` creates the new `Account` object (RealSubject) using `proceed()` (line 28), which will be added to the pending RealSubjects list (`rs`), i.e. the reference to return to the Proxy that is going (line 30) to be reflectively created. In this scenario the `Checker` (Proxy) constructor will try to instantiate its own RealSubject, so a second `trapCalls`-related advice will be triggered putting on hold the previous one, interrupting it while waiting for the completion of `newInstance()` in line 30. In this second `trapCalls` execution the `isCallerProxy()` call returns `true`. The if statement on line 2 (figure 3.16) is false, as the `proxies` map has not yet been populated with the instance of `Checker` (Proxy) being created, however, after adding to the `classes` map the pair, e.g.,

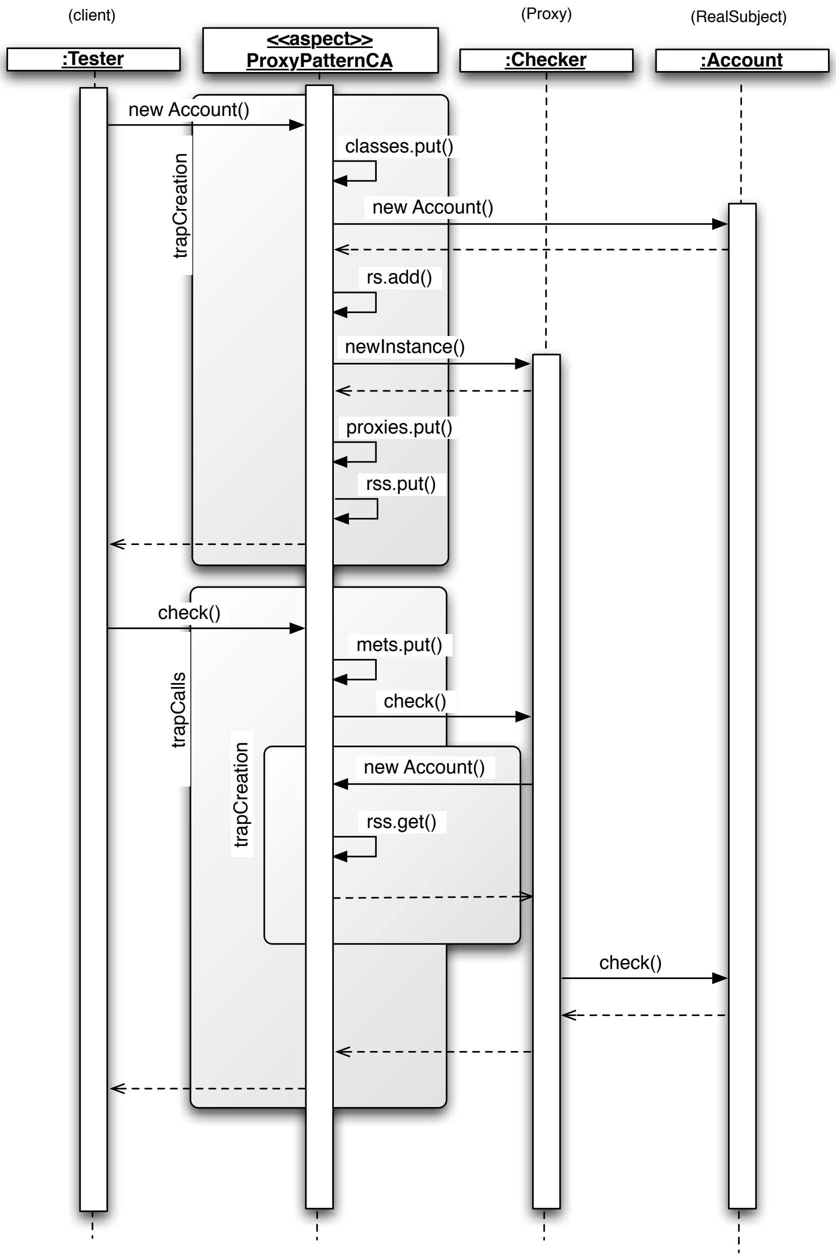

Figure 3.18: *A* Proxy *creating its own* RealSubject *outside its own constructor*

(`Checker@255`, `"Checker"`), the return statement on line 6 evaluates to `true`, as the class name found in the annotation (lines 21–22, figure 3.15) of the `Account` class (i.e. `"Checker"`) is the same as the Proxy one (i.e., again, `"Checker"`). The `trapCreation` advice's execution continues with the conditional in line 24, which returns `rs.peek()` to the `Checker` object, i.e. the reference to already created RealSubject. This ends the second `trapCreation` execution, returning the control to the first one, which was interrupted on line 30. Now the `p` object is finally assigned, holding the reference to the new reflectively created Proxy (a Checker), and the proxies and rss maps are populated, with pairs such as, respectively, (Account@551, Checker@255) and (Checker@255, Account@551). The last instruction (line 34) removes the reference to the RealSubject (an Account) held in rs, returning it to the original client, Tester, which requested the creation.

A different scenario takes place when a Proxy does not create its RealSubject inside its own constructor, but inside another method (figure 3.18). In this scenario everything remains the same as in the previous one until the reflective call creates the new Checker (Proxy) on line 30, figure 3.15. As the Checker's constructor does not create its RealSubject the trapCreation advice ends after putting the new (Proxy, RealSubject) pair in both proxies and rss maps, hence without the double activation of the advice previously described.

Now suppose that the Checker class is programmed to instantiate an Account object as its RealSubject in the check() method, and the same Tester client invokes the check() method on the Account reference obtained by the ProxyAspectCA, this activates the trapCalls advice. Since the calling object is a Tester (thus neither a Proxy nor a RealSubject) the advice has to invoke a check method on the related Proxy, by means of the invokeOnProxy() method (line 14, figure 3.15), after caching the results of the reflective operations (lines 14–15, figure 3.16). The Proxy will have to create its RealSubject inside this method since it had not been created before, this triggers the trapCreation advice, note that the trapCalls advice was waiting for the end of the check() method. The trapCreation advice will identify the caller as a Proxy (isCallerProxy() yields true as proxies.containsValue() is true due to the previous addition of the reference to this very Proxy), and instead of letting the Proxy (Checker) create another RealSubject, advice trapCreation will return its paired RealSubject stored in the rss map. The Proxy (Checker) now has a reference to its Proxy and will (and can) invoke the original check() method on its RealSubject.

Table 3.2 shows sample pairs' values[3] from the proxies, classes and rss maps after the execution of the following sample code where a Tester class creates an Account object as in the previous description

[3] The values shown have to be read as Java object references.

proxies		classes		rss	
RealSubject	Proxy	`Object`	`String`	Proxy	RealSubject
Account@551	Checker@255	Tester@739 Checker@255	`"Tester"` `"Checker"`	Checker@255	Account@551

Table 3.2: *Sample values for the* ProxyPatternCA*'s maps*

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Bypass {
    String []  value ();
}
```

Figure 3.19: *Annotation for the* @Bypass *annotation*

```
Account acc = new Account();
acc.check();
```

3.3.4 `ProxyPattern` aspect's variants

Both the AA and CA versions of the *Proxy* design pattern might be used in scenarios, such as *smart reference* or *protection proxy*, however an additional variation that is made available is a *bypassable Proxy*, that is the possibility for selected (privileged) clients to directly access a RealSubject bypassing the Proxy shield. This might be considered a violation of the principle of the Proxy design pattern, however it gives to the programmer the capability to (possibly) choose some classes to be ignored by the Proxy enforcement mechanism implemented by the aspect. As the rest of the approach, the granularity remains at the class level, i.e. a class might be in the *bypass whitelist*, but a specific object can not.

The `@Bypass` annotation is shown in figure 3.19 and can be used in combination with the `@Proxy` one, for example as in the following code

```
@Bypass({"Bank", "ATM"}) @Proxy("Checker")
public class Account { ...  }
```

where the same `Account` class of the previous examples is declared to be a RealSubject with an automatically created Proxy (a `Checker` object) and the classes `Bank` and `ATM` will not be affected by the aspect and can access any `Account` objects without being redirected to a `Checker`.

```
1 private WeakHashMap<Object, String> exclude = new WeakHashMap<Object, String>();
2
3 private boolean isCallerExcluded(Object t , String s) {
4     if (exclude.containsKey(t))
5         return exclude.get(t).equals(s );
6     String c = t.getClass (). getName();
7     exclude.put(t,c );
8     return c.equals(s );
9 }
```

Figure 3.20: *Additional code to manage the* @Bypass *annotation*

The existing `ProxyPattern`'s aspects need to be enhanced with the code shown in figure 3.20. Please note that for the sake of simplicity and comprehensibility the shown code just refers to the case of a bypass list holding a single element; however the code for the case of a list is logically equivalent and similarly managed, with additional code used to recover and check the full list.

This utility method takes two parameters, the first one, `t`, is the reference to the calling object, the last is a string representing the value of the `@Bypass` annotation, i.e. the name of a class authorised to access that specific RealSubject directly.

The `exclude` map holds the pairs (object, class name) of all objects that invoke an intercepted method. It is populated in the same way as the `isCallerProxy()` method (figure 3.16), so in the CA case the code can be modified to use only the `classes` map, as they serve the same purpose.

To check whether or not `t` is allowed to access the RealSubject its class name is compared with the class name found in the `@Bypass` annotation. The class name is obtained either by accessing the `exclude` map (line 4, figure 3.20) or using reflection (line 6, figure 3.20).

The `isCallerExcluded()` method has to be inserted into the `trapCalls` and `trapCreation` advices, to let a privileged client to behave such as a Proxy or a RealSubject would do, i.e. having its calls not redirected by the aspect. The `trapCalls` advice (lines 12–15, figure 3.15) is enhanced with another check for the condition, as shown in figure 3.21, where the caller is managed in the same way as a Proxy or a RealSubject.

For the `trapCreation` advice the same behaviour must be implemented. In this case the privileged class can not be managed in the same way as, e.g., a RealSubject.

```
Object around(Object o, Object t , Proxy a): trapCalls(o, t , a) {
    if ((t == o) || (t == proxies.get(o)) || (isCallerExcluded(t, a.value())) )
        return proceed(o, t, a);
  return invokeOnProxy(thisJoinPoint, a, proxies.get(o));
}
```

Figure 3.21: *Additional code to manage the* @Bypass *annotation*

```
if (isCallerExcluded(t, ((Bypass) thisJoinPoint.getSignature()
            .getDeclaringtype().getAnnotation(Bypass.class)).value())
   return proceed(t);
```

Figure 3.22: *Additional code to manage the* @Bypass *annotation*

The only change that need to be done is the addition of the conditional statement in figure 3.22 as the second instruction of the advice, before line 23 of figure 3.15, to allow the execution of the intercepted join point.

Another variation that can be implemented is to support of a sequence of proxies, e.g. when an `Account` class plays as a RealSubject for a `Checker` class (a Proxy), and the `Checker` class also plays the RealSubject role for a `Counter` class. It is possible to extend the `ProxyPattern` aspect with additional pointcuts and advices to accommodate such a scenario.

The annotation to use remains the same, and would be used, as expected, on both RealSubject classes, i.e.

```
@Proxy("Checker") class Account { ... }
@Proxy("Counter") class Checker { ... }
```

To automatically capture the calls on the `Checker`'s methods an additional pointcut has to be defined. The existing one can not be directly used as it would not capture the dynamic invocation (line 15, figure 3.12) as it is not a call to a Proxy but to the `invoke()` method of the `Method` class, which will invoke a Proxy method, however not as an exposed join point. Thus an additional `trapReflectiveCalls` pointcut captures any `Method.invoke()` and checks if the captured call is directed to a class marked with the `@Proxy` annotation, the rest of the related advice is essentially the same as the non-reflective variant already described.

3.3.5 Evaluations

The described `ProxyPattern` aspects present a possible puzzling feature, some might define it inversion of control, however, in practice, this is not the case. Using the `ProxyPattern` it seems as if the usual flow of execution when using a Proxy is changed as a client (apparently) directly creates a RealSubject object and invokes its methods. Albeit this is exactly what the proposed AODP prescribes the programmer to do, the final, observable behaviour when using the `ProxyPattern` is the same as the object-oriented alternative. E.g. if the `Account` class is playing the RealSubject role, the invocation of the `check()` method on an `Account` object will be intercepted, i.e. interrupted, by the aspect and will not be executed unless its related Proxy (the paired `Checker` object) logic allows the execution, exactly as it happens in the object-oriented version of the pattern. The intent of the *Proxy* design pattern is fully obeyed. It might also be worth noting that this same behaviour is observable in the aspect version in [HK02].

Another characteristic of the design of the `ProxyPattern` aspect is how it treats the possibile (RealSubject, Proxy) pairs with a granularity set at the class level, i.e. once the `Account` class is associated to the `Checker` one as its Proxy, all the `Account` instances will inherit this association, so a single, selected instance can not be associated to, say, a `CounterProxy` class. Managing such a case would still be possibile however at the cost of a considerably more complex aspect.

Lastly, the pointcuts capturing the methods' calls to be proxied assume that the call for such methods happens outside a static method, because the `this` construct (line 6, figure 3.12) can not capture an invocation happening from a static scope, as no `this` object would be associated to it. This can be easily solved using a variation of the pointcut which captures the execution of the method and instead of relying on the `this` construct, using the `thisJoinPoint.getThis()` method to initialise the `t` variable.

3.4 *Observer*

The *Observer* design pattern is used to allow loose coupling in the observation relationship between objects, i.e. when objects of (possibly) different classes (playing the ConcreteObserver role) are interested in changes of states of another object (playing

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

Figure 3.23: *Annotation for the* Observer *design pattern*

the ConcreteSubject role).

It proposes to make the ConcreteObservers implement the *Observer* interface, so to deal with different ConcreteObservers in the same fashion, and to make a ConcreteSubject inherit from a *Subject* class. The *Subject* class contains the code for the management of the list of Observers (with the `attach()` and `detach()` methods) and the `notify()` method is used to inform all the ConcreteObservers of a state change in the ConcreteSubject.

When a ConcreteSubject changes its state, it will invoke the `notify()` method to let all the ConcreteObservers know about it, and in turn all the ConcreteObservers will invoke their `update()` method to get the new state from the `ConcreteSubject`.

This architectural detail effectively takes the Observers-related code out of a ConcreteSubject class, however, as the `notify()` invocation has to be explicitly performed by the ConcreteSubject, it makes it tightly coupled with its `Subject` superclass. Apart from the coupling, such a call is not part of the main responsibility of a class, as it is added just to implement the design pattern.

The Subject–ConcreteSubject class relationship can become a burden to deal with in an object-oriented language, such as Java, where multiple inheritance is disallowed, as a ConcreteSubject has to inherit from Subject and can not inherit from other classes[4].

Another disadvantage of the coupling that comes with the object-oriented solution is that when a ConcreteSubject has to be used in an architecture where it does not have to play this role anymore, all the `notify()` calls have to be removed from the code.

[4]Unless the designer uses composition to provide the Subject behaviour.

```
public class Account {
    double balance=0; ...

@Observer(clas="Store", meth="update", par="balance")
public void deposit(double i) {
    balance+=i;
}
```

Figure 3.24: *Sample usage of the* @Observer *annotation*

3.4.1 Aspect-oriented and annotated *Observer*

The ObserverPattern aspect (figure 3.25) encapsulates all the behaviour needed to handle the Observers list. By capturing any method call marked with the @Observer annotation (figure 3.23), the aspect will automatically notify any Observer after the observed method has been executed successfully, so a ConcreteSubject need not (i) be subject to a hierarchy constraint, i.e. extend the Subject superclass, (ii) intertwine its own domain code with interspersed calls to its Subject superclass. Also there is no need for a ConcreteSubject class to implement the code for the Observers' list management and notifications, as it will be the ObserverPattern that will take care of them (verifying *SoC*), with the aspect fully implementing the behaviour of the Subject role (verifying *ECoR*).

The annotation is intended to be used on any class' method whose results are of any interest for a ConcreteObserver. As any method of any class can be annotated, the aspect is general and reusable in any application: to impose (remove) an observation relationship between classes it is sufficient to weave (remove) the ObserverPattern in the application without having to also change other classes (fulfilling *SPoC*).

The parameters of the annotation are meant to be used on methods of a ConcreteSubject as in the example in figure 3.24 where a deposit() method is declared to be observed, hence its declaring class (Account) plays the ConcreteSubject role. The retention policy of the annotation is declared to be "runtime" to allow the parameters to be reflectively read then.

In particular, every time the deposit() method ends (without launching any exception) the update() method declared in the Store class will be invoked, using the fresh value for the balance variable as a parameter, on any ConcreteObserver

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

  pointcut trapCalls(Observer ann, Object obj): this(obj) &&
    execution(@Observer * *.*(..)) && @annotation(ann);

  after(Observer ann, Object obj): trapCalls(ann, obj) {
    try {
      Class c = Class.forName(ann.clas());
      Method m = c.getMethod(ann.meth(), new Class[]{Object.class});
      List observers = subjects.get(obj);
      Field par = obj.getClass().getDeclaredField(ann.par());
      Object pp = par.get(obj);
      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> obsLst = subjects.get(subj);
    if (obsLst == null) obsLst = new LinkedList<Object>();
    obsLst.add(obs);
    subjects.put(subj, obsLst);
  }
}
```

Figure 3.25: *The* ObserverPattern *aspect*

(of `Store` class) that has previously been attached to the Observers list.

As the `ObserverPattern` aspect automatically takes care of notifying the interested Observers, there is no possibility for the programmer to make mistakes such as forgetting to invoke a `notify()` in the ConcreteSubject code after some value has been updated, thus fulfilling the *REoR* property.

3.4.2 ObserverPattern aspect

The ObserverPattern (figure 3.25) implements the behaviour of the Subject role. The main behaviour is obtained by a single pointcut (trapCalls) and its related advice.

As the aspect has to manage the list of Observers, it uses the subjects map to store it in a general way, i.e. unconstrained by the actual classes involved. The subjects map holds ConcreteSubjects as keys, to which a list of ConcreteObservers are associated. Both roles are stored as generic object references, to let the aspect be completely independent of the classes it will work with, any needed information about the actual class will be discovered at runtime using reflection. The map is declared as a WeakHashMap, to allow the garbage collector to remove entries whose keys are not referenced. This is considered reasonable as if a ConcreteSubject is no longer in existence it should not have any ConcreteObservers attached.

To populate the subjects map an addObserver() method is provided; this is used by client classes to dynamically register a ConcreteObserver as observing a ConcreteSubject, as in the standard object-oriented implementation. A similar method to dynamically remove a ConcreteObserver is provided, but not shown in figure.

To add a new ConcreteObserver in the list, a client just needs to invoke the method as follows

```
ObserverPattern.aspectOf().addObserver(cs, co);
```

where cs is an instance of a ConcreteSubject, co an instance of a ConcreteObserver and at least a method of cs is annotated with the class of co as the clas parameter of the @Observer annotation.

The after construct available within AOP fits as a natural way to implement the fundamental behaviour of the *Observer*, i.e. intervening *after* a method updating the state of the ConcreteSubject has been executed, to update all ConcreteObservers.

The trapCalls pointcut intercepts all the executions of any method annotated with the @Observer annotation, also capturing a reference to the caller of the method, using construct this, and a reference to the annotation, to be accessible in its related advice.

The advice has to invoke on any ConcreteObservers the method specified in the annotation. To obtain a reference to the update method (whose execution has been

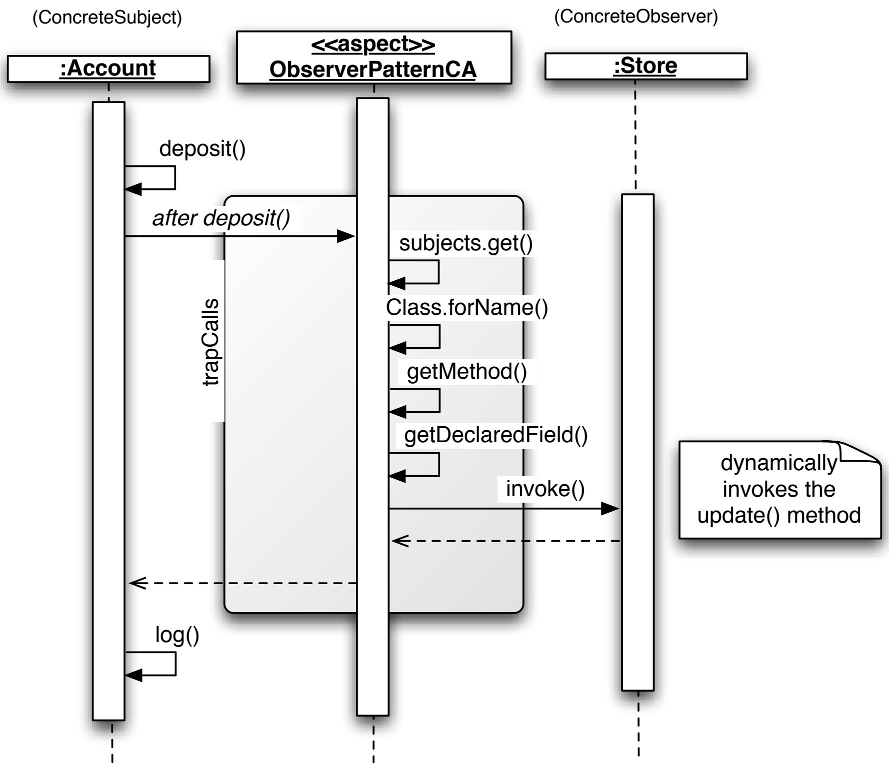

Figure 3.26: *Automatic activation of the* ObserverPattern *aspect*

intercepted by the aspect) all the parameters of the annotation are needed. In line 9 the class found as the `clas` parameter of the annotation is retrieved in `c`. The name of the method to be executed for the update, retrieved in `m`, is found in the `meth` parameter of the annotation and is used in line 10. In line 12 the parameter (`ann.par`) read from the annotation is retrieved as a `Field` and then used to retrieve its current value in `pp`, so to pass it as the updated value to the ConcreteObservers.

The list of all ConcreteObservers associated to the ConcreteSubject reference (`obj`) is retrieved in line 11, and used for the loop (lines 15–17). All the ConcreteObservers are updated by the dynamic invocation (line 17) of their `m` method, using `pp` as the value for the parameter.

Supposing the `Account` class annotated as in figure 3.24, and the existing objects `a` and `o` respectively of class `Account` and `Store`, the following (simplified) code snippet

```
ObserverPattern.aspectOf().addObserver(a, o);
a.deposit(7.39);
logObject.log(time+":  new value deposited.");
```

when executed, becomes as if it were written

```
ObserverPattern.aspectOf().addObserver(a, o);
a.deposit(7.39);
o.update(7.39);
logObject.log(time+":  new value deposited.");
```

as also shown in the sequence diagram in figure 3.26.

3.4.3 `ObserverPatternCA` aspect

The `ObserverPatternCA` (figure 3.27) is an enhanced version of the regular `ObserverPattern`. Since the latter uses many reflective calls to be general, and the results of their invocations remain constant when applied to the same input arguments, it is possible to use a map to store those results instead of making the same computations every time the `trapCalls` advice is activated. This enhancement is just for the running times of the advice's execution, as both versions are functionally equivalent, they also they share the same pointcut. The changes are only related to the advice and the addition of the `ref` map as a caching device.

The cached values are stored using a `reflectInfo` class (figure 3.28), to store the interested values, i.e. the reference to the class of the ConcreteObserver, the method to call and its parameters, so to avoid all the repeated reflective instructions.

3.4.4 Evaluations

An interesting side effect of the proposed *Observer*'s approach is that different methods of the same `ConcreteSubject` class can be independently observed by instances of distinct `ConcreteObserver` 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 those of the annotation marking another method. This would allow alerting different methods and possibly different

```
public aspect ObserverPatternCA {

   private WeakHashMap<Object, List> subjects = new WeakHashMap<Object, List>();
   private WeakHashMap<Signature, reflectinfo> ref =
            new WeakHashMap<Signature,reflectinfo>();

   pointcut obser(Observer ann, Object obj): this(obj) &&
      execution(@Observer * *.*(..)) && @annotation(ann);

   after(Observer ann, Object obj): obser(ann, obj) {
      try {
         Signature sig=thisJoinPoint.getSignature();
         Class c=null; Method m=null; Field par=null;
         List observers=null;
         Object pp=null;
         if (! ref.containsKey(sig)){
            c = Class.forName(ann.clas());
            m = c.getMethod(ann.meth(), new Class[] { Object.class });
            observers = subjects.get(obj);
            par = obj.getClass().getDeclaredField(ann.par());
            pp = par.get(obj);
            ref.put(sig, new reflectinfo(c, m, par));
         }
         else{
            c=ref.get(sig).getClas();
            m=ref.get(sig).getMethod();
            observers = subjects.get(obj);
            par=ref.get(sig).getField();
            pp = par.get(obj);
         }
         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) { /* ... */ }
   }
}
```

Figure 3.27: *The* ObserverPatternCA *aspect*

```
public class reflectinfo {
  Class c;
  Method m;
  Field par;

   reflectinfo (Class c, Method m, Field par){
    this.c=c;
    this.m=m;
    this.par=par;
  }

  Class getClas(){ return c; }

  Method getMethod(){ return m; }

  Field getField(){ return par; }
}
```

Figure 3.28: *The* reflectinfo *class*

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.

The `ObserverPattern` shown in figure 3.25 manages only instances of a class where only instances of one class playing as ConcreteObserver role have to be notified for each method, the extended version handling a list of classes, using

```
@Observer(clas={"Store", "View"}, meth="update", par="balance")
```

is not shown for the sake of brevity.

3.5 *Composite*

The *Composite* design pattern aims at making the interaction of clients' classes the same, both when interacting with simple objects or with aggregates of objects. The intention is to avoid clients' classes to implement different behaviours when interacting with the two possible cases.

In the suggested solution of [GHJV94] the Component interface (or abstract

```
public @interface Composite { }
```

Figure 3.29: *Annotation for the* Composite *design pattern*

class) is a parent of both Leaf and Composite, and defines methods that these children have to implement, each relative to its own nature. A Leaf class is a simple class, thus implementing an `operation()` method acting by itself, while a Composite class represents an aggregation, i.e. it may contain both Leafs and Composites, thus implementing the same `operation()` method acting upon all the subtrees starting from the Composite node it represents.

For example, a file system might be modeled using a Leaf class to represent a file in a file system, and the Composite one to represent a directory. When a client calls the `size()` method on a Component it will receive the size of the file (if the Component is a Leaf) or the size of the files in the subdirectory starting from the Component (if it is a Composite).

In the so-called safe solution, the Composite class has to provide methods for the addition (or removal) of its children elements, thus having to cope with references to such elements. In the so-called transparent solution these methods are implemented in the Component class. Thus a Composite class is expected to mix its own domain code with the code needed to implement the handling of the aggregated objects. This mixing hinders its reusability, as the class becomes tightly coupled with the role it is playing.

3.5.1 Aspect-oriented and annotated *Composite*

The `CompositePattern` aspect shown in figure 3.30 is used in concert with the `@Composite` annotation (figure 3.29). Such annotation acts just as a tagging annotation, as it bears no parameters, it just marks a class as playing the Composite role. It has been defined with "runtime" retention policy as its existence has to be detected while the application is running.

The aspect intercepts any method call (say `m()`) to a Composite object (say of class `C`), and handles the expected aggregation behaviour for `m()` by reflectively invoking it on all the children of `C` and finally on the Composite it was originally directed to. This renders the code of the `C` class effectively free from the non-domain code it should have implemented, as this code is kept in the `CompositePattern` aspect (*SoC*). Moreover, `C` does not need to implement additional methods such as `add()` and `remove()` to manage its children, as these are provided by the aspect. Since all the DP behaviour is encapsulated in the `CompositePattern` aspect, the *ECoR* criterion is verified.

To impose (or remove) the Composite role to a class it is sufficient to just add (or remove) the `CompositePattern` aspect when compiling the application, no other parts of the application have to be changed (*SPoC*). Any client class invoking the `m()` method also remains the same, when `C` stops playing the Component role. In any case, the programmer can not err by invoking the wrong method on (*REoR*).

3.5.2 `CompositePattern` aspect

`CompositePattern` in figure 3.30 consists of one pointcut (`trapOperations`), its related advice and several utility methods that implement the main Composite behaviour.

The `comps` map holds the children list of any Composite object regardless of the actual class, as it is defined to hold a generic object as a key and a list of generic objects as its children's list. This is necessary to keep the aspect completely general and independent of the application it is woven into. The utility method to manage the `comps` map shown in figure are used to add a child to an existing Composite object, and to get the list of all the children of a given Composite.

The `trapOperations`-related pointcut carefully selects all the calls to any method declared in a class marked with the `@Composite` annotation. It captures a reference to the Composite object to which the call is directed.

The advice starts by retrieving all the children of the intercepted Composite (`co`) then it reflectively detects the name of the intercepted method (line 11), `mc`. The `mc` method is then invoked on every children of `co`.

The invocation takes place via the local `invokeOnChild()` method. This accepts the reference to the child, extracts method (`m`) with the same name and parameters (line 21) of the initially intercepted method on `co` and then dynamically invokes `m` on the child, using the (possible) actual parameters collected from the context using `getArgs()` on `thisJoinPoint`. The return value obtained from the invocation is returned in `res`.

After the invocation of `mc` on a child, the result is passed to the Composite

```
public aspect CompositePattern {
   private Map<Object, List> comps = new Hashtable<Object, List>();

   pointcut trapOperations(Object co) :
       call(* *.*(..)) && target(co) && @target(Composite) && !within(CompositePattern);

   Object around(Object co) : trapOperations(co) {
      List children = comps.get(co);
      if (children != null) {
         Object res;
         Method mc = ((MethodSignature) thisJoinPoint.getSignature()).getMethod();
         for (int i = 0; i < children.size(); i++) {
            res = invokeOnChild(thisJoinPoint, mc, children.get(i));
            invokeOnComposite(mc, co, res);
         }
      }
      return proceed(co);
   }

   private Object invokeOnChild(JoinPoint jp, Method mc, Object ch) {
      try { Method m = ch.getClass().getMethod(mc.getName(), mc.getParameterTypes());
            return m.invoke(ch, jp.getArgs());
      } catch (Exception e) { /* ... */ return null; }
   }
   private void invokeOnComposite(Method mc, Object co, Object res) {
        try { Method mce = co.getClass().getMethod(mc.getName(), res.getClass());
                 mce.invoke(co, res);
      } catch (Exception e) { /* ... */ }
   }
   public void addChild(Object comp, Object child) {
      List childLst = comps.get(comp);
      if (childLst == null) childLst = new LinkedList<Object>();
      childLst.add(child);
      if (childLst.size() == 1) comps.put(comp, childLst);
   }
   public List getChildrenList(Object comp) {
      return comps.get(comp);
   }
}
```

Figure 3.30: *The* CompositePattern *aspect*

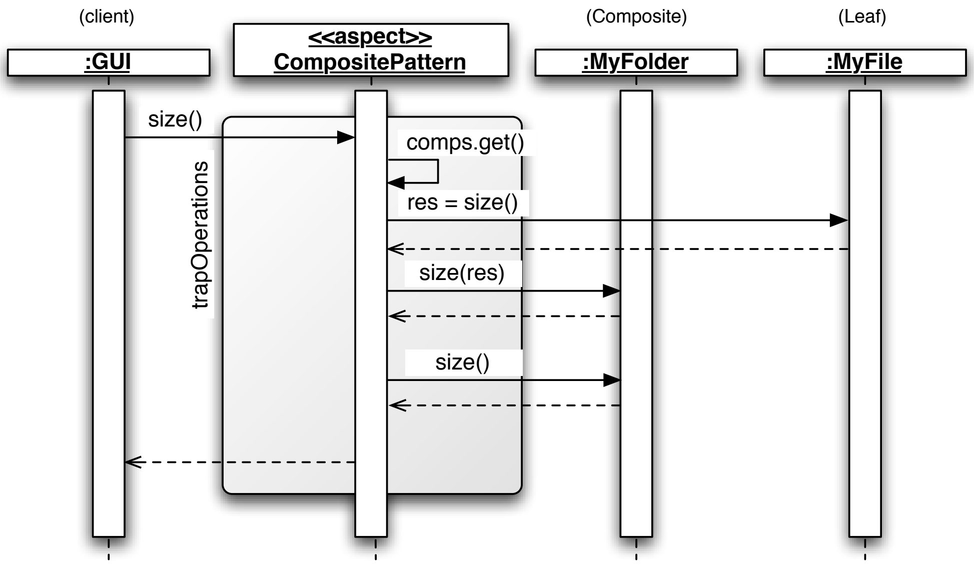

Figure 3.31: *A possible scenario for the* CompositePattern *aspect*

(`co`), using the `invokeOnComposite()` method. The programmer has to provide the Composite class with a method with the same name as the captured one (held in `mc`) but with an additional argument (say `mce`). The responsibility of `mce` is to collect and operate on all the partial results obtained by the execution of `mc` on every single children. So the type of the extra argument must be the same of the result type of `mc`. The execution of the `invokeOnComposite()` method takes place by means of dynamic invocation (line 27).

As a note, the `mce` method might seem a further responsibility that is added on the Composite class, however on the contrary it can easily be considered a responsibility of a Composite class, i.e. a class that manages and manipulates instances of its peers and so has all the knowledge to handle the intermediate results.

After the `mc` method has been executed on every children of `co`, the `proceed()` statement (line 17) executes it on the original `co` object on which the call had been performed.

Figure 3.31 shows a scenario with a `GUI` object invoking a method to compute the size of a `MyFolder` object. The aspect takes care of dynamically invoking the `size()` method on the child of the `MyLeaf` object, then, the `size()` method (with the additional parameter) is invoked on the `MyFolder` object. Eventually the `size()` method is invoked to get the final result. All the invocations performed by the aspect, `comps.get()` excluded, are dynamic invocations.

```
pointcut trapRecursiveOperations(Method co, Object comp, Object[] pars) :
    call(* Method.invoke(..)) && target(co) && args(comp, pars)
    && within(CompositePattern)
    && if(co.getDeclaringClass().getAnnotation(Composite.class)!=null)
    && !cflow(call(void invokeOnComposite(..)));

Object around(Method m, Object comp, Object[] pars) :
    trapRecursiveOperations(m, comp, pars) {
        List children = comps.get(comp);
        Object res;
        if (children != null) {
            for (int i = 0; i < children.size(); i++) {
                try { // invoke on child, then on composite
                    Method mc = children.get(i).getClass().getMethod(m.getName(),
                                                        m.getParameterTypes());
                    res= mc.invoke(children.get(i), pars);
                    invokeOnComposite(m , comp, res);
                } catch (Exception e) { /* . */ return null; }
            }
        }
        return proceed(m,comp, pars);
    }
```

Figure 3.32: *Recursive variation for the* CompositePattern *aspect*

The `CompositePattern` shown in figure 3.30 is a simplified version of the full one, the former being shown to understand its basic internal behaviour. The shown version can not capture reflective calls to the `mc` method, thus if a visited Component (line 13) instead of being a Leaf is a Composite, it will not be visited recursively. To avoid this behaviour the aspect has to be enhanced with the additional `trapRecursiveOperations` pointcut, and its related advice, shown in figure 3.32.

The basic behaviour is the same as the original `trapOperations` pointcut, however the `trapRecursiveOperations` is more complex. The original `trapOperations` pointcut just captures all the calls to a method from a class marked with the `@Composite` annotation, however this pointcut will not be triggered when the `invokeOnChild` method is reflectively invoked (line 22, figure 3.30), as the only exposed join point is the `invoke()` method of the `Method` class, not the method that will be dynamically called. So to capture this dynamic invocation the call to be intercepted is

```
pointcut trapOperations(Object o) :
  call(* *.*(..)) && target(o) && @target(Composite)
  && !within(CompositePatternCA) && !within(CacheAO);

private Object invokeOnChild(JoinPoint jp, Object ch) {
   Method m = CacheAO.getM(ch, jp);
   try {
      return m.invoke(ch, jp.getArgs());
   } catch (Exception e) { /* ... */ }
   return null;
}

private void invokeOnComposite(JoinPoint jp, Object c, Object r) {
   Method m = CacheAO.getMco(c, r, jp);
   try {
      m.invoke(c, r);
   } catch (Exception e) { /* ... */ }
}
```

Figure 3.33: *Variations for the* CompositePatternCA *aspect*

exactly `Method.invoke()`, this, however, has to occur inside the `CompositePattern` aspect (line 3, figure 3.32). Moreover the invocation has to occur outside the control flow of the `invokeOnComposite()` method, so to avoid infinite recursion (line 5) and the target class, i.e. the `co` object, has to be marked with the `@Composite` annotation. The related advice behaves essentially as the original one, with trivial differences in getting the `mc` method.

3.5.3 `CompositePatternCA` aspect

Figure 3.33 shows the `CompositePatternCA`, i.e. the functionally equivalent, faster, version using a caching device. The cache maps in the utility class `CacheAO` (not shown) a reference to the intercepted method, i.e. the method to invoke on all the children of Composite and the method with the same name with the additional parameter (see section 3.5.1), indexing them by the `SourceLocation` of the intercepted join point.

3.5.4 Evaluations

A simple variant of the `CompositePattern` allows the programmer to choose whether to enable automatic calls of methods on children or just let the aspect take care of the children's list. This allows the Composite code to incorporate any algorithm that performs any desired action, say selectively call the provided operation on the children, use the intermediate results, etc. It is worth noting that a Composite can access the children's list using the utility methods provided by the aspect. The corresponding aspect is not shown as it is a subset of the one in figure 3.30.

3.6 Analysis and other design patterns

All the presented AODPs fulfil the properties stated in section 1.1, such as, emphasizing one of their main strong points, their complete reusability without any changes, just as a library function. This is different from the other approaches found in literature, examined in chapter 6, as the reasonably clever use of reflection renders unprofitable the counterpart implementation of abstract and concrete aspects.

A tradeoff brought by the generality of the AODPs is about the internal structures (maps) of the aspects that, being generic, can not assure the type safety of held objects. Thus, while an aspect can accommodate any class in its maps, it may also be subject to runtime exceptions as such classes are obtained as strings from the annotations' parameters. Such strings are however part of the source code and do not change, as they are not obtained from the user's input but inserted once and for all by the programmer, therefore an appropriate test suite can be used to verify the correctness of a complete application using the proposed patterns' implementations.

Another limit bound to the technology of choice, AOP in this case, might be argued to be the comprehensibility of the (flow of the) final application. It might be more difficult to understand the flow of a program without tools, such as IDEs, which remind the programmer if an instruction is *adviced* by an aspect, or to understand if there is any interaction between the aspects. Again, this is an issue that comes with AOP, not brought by the proposed AODPs' implementations. However, also the opposite might be argued: since a class is explicitly marked with a simple (if not self-explanatory) annotation[5], such a class makes it clear its additional role

[5]Unless the programmer chooses to use the alternative described in section 3.7.

and so its expected (superimposed) behaviour. E.g. even when the programmer edits a source file with a simple text editor a method marked with the `@Observer` annotation should make clear for the programmer that the method execution will be followed by an update to its observers.

The proposed AODPs are implemented in Java and AspectJ, however the approach should be general enough to be ported to any object-oriented language supporting the same basic tools used (aspects, reflection and annotations).

The detailed AODPs presented in this chapter are just a subset of some of the most basic and used design patterns described in literature [GHJV94, BMR+96, SSRB00]. Unfortunately this novel aspect-oriented implementation is not automatically extendable to arbitrary patterns without human intervention.

A useful discrimination made in [HK02] is between a *defining* and a *superimposing* role for a design pattern. The former is a role that "defines the participants completely", i.e. played by a class decoupled from the whole system except for the pattern-related classes (such as the FlyweightFactory role), while the latter is a role which adds responsibilities to a system class to implement the pattern's behaviour (such as the Flyweight role). The definition is however not a strict one. However, using this loose partition it is easy to see how the proposed AODPs encapsulate the code of the superimposing roles of the original object-oriented patterns as advices and methods in the AODPs' aspects, as both *ECoR* and *SoC* properties verified by the AODPs imply that an application class playing a role should not be concerned with the code for such a role's implementation.

It is however possible to extend the approach to implement other design patterns in the aspect-oriented fashion described in this dissertation, although not in a trivial, automatic, way. The main difficulty encountered for such an extension is to write an aspect that can be completely reusable *as is*, capable of supporting all the properties described in section 1.1. The definition of one or more additional annotations to describe a design pattern might be relatively a simple task, however the non-trivial part of the approach is the definition of general pointcuts for *any* application, to be coupled with advices that implement (add to the application) the additional non-domain behaviour for the superimposing roles of the pattern. This problem can be separated in two different but strongly related subproblems that the programmer has to tackle. The first is the very definition of the pointcuts as hook points to any application, in terms of AOP constructs, the second one is, given a context collected by the defined pointcuts, how to provide an advice's code with enough context information to be able to perform its own expected behaviour. In approaches such as [HK02] the definition of a design pattern is not as problematic, as the partition the authors make between an abstract and a concrete aspect allow them to define just an abstract pointcut to be concretised in the concrete aspect, at the price of limiting the reusability of their code, as the concrete aspect, and especially its pointcuts, has to be specifically written for *any* instance of the pattern.

Other AODPs' implementations have been analysed and sketched, as reported in the following.

The *Abstract Factory* and *Factory Method* use a parameterised `@Product` annotation as a way to mark the ConcreteProduct, used to discover all the implementations of the Product interface, as the parameter specifies exactly the Product class. The provided aspect intercepts the creation of all the instances of a class marked as Product and decides which class to instantiate, among subclasses of the Product interface. The created instance is returned to the client. The aspect enforces the creation logic for any ConcreteProduct class (*REoR*), on the contrary the object-oriented version does not help the programmer for possible wrong uses of a `new` call on a class which should be managed by a factory.

The Mediator design pattern is implemented using an annotation on a subset of the methods of a ConcreteColleague class. The aspect intervenes after such methods have been successfully executed to call methods on other classes also playing as ConcreteColleagues. The aspect connects all the ConcreteColleagues, however leaving the code of classes independent of each other. The used annotation indicates the name of classes playing as ConcreteColleagues, and in addition the name of the method to be called. The aspect takes care of collecting the references to instances of classes playing as ConcreteColleagues.

The *Memento* design pattern uses the `@Memento` annotation to mark a class playing the Originator role, the aspect takes care of keeping a copy of the state of any Originator object. The copy of the state is performed before any execution of a method of an Originator class. The use of reflection permits to perform the copy of the state of the object without knowing beforehand its structure, as the object is introspected at runtime accessing its fields and respective values. A saved state for an object can be restored using a provided method. The backup of the state can be dynamically enabled or disabled using provided methods.

3.7 Annotations' connector aspect

To be able to use the AA and CA versions of the presented design patterns, the programmer needs to explicitly define the roles to be played by the involved classes using the provided annotations. While annotations provide a means to easily and concisely superimpose design patterns' roles onto application classes, it would be still more desirable for the latter to be even *annotation-free*. This can be achieved by means of a connector aspect, designed to annotate classes as necessary.

As a crosscutting instruction, AspectJ provides the `declare` instruction to inject static changes to a class. An example of such an aspect, to be used with the `ProxyPattern` aspect, is the following:

```
public aspect Connector {
  declare @type: Account: @Proxy("Checker");
}
```

where the classes `Account` and `Checker` are forced to act as, respectively, a RealSubject and a Proxy (as in figure 3.10). Once both the connector and the `ProxyPattern` aspect are woven into the application, the `ProxyPattern` aspect will have the necessary hooks to intervene at runtime.

Such a connector aspect is application-dependent, but the dependency is actually a simple parameterisation on class names, the connector structure being fully generic. The use of such a connector aspect would improve the *SoC* criterion, leaving untouched the involved class even from the annotation.

However, both options (direct annotation and connector aspect) are viable and of practical use in different scenarios. For example, in the *Proxy* pattern both the RealSubject and Proxy can change their names during development for evolution purposes. Using the sample architecture of figure 3.10, where the `Account` class (a RealSubject) must be changed to `BankAccount`, if the annotation is directly written into the class source file, then no change to the annotation would be necessary, as the annotation makes no reference to the class name on which it is applied. Instead, using the connector aspect, the connector aspect itself must be changed to adapt it to the new class name in its `@type` declaration[6].

However, a quite opposite scenario could also happen. If a Proxy class name has to be changed, e.g. `Checker` becomes `AccountChecker`, when using the annotation

[6] Such changes can be considered aspect-aware refactorings [HOU03, IZ03].

on the class file of the RealSubject, *all* the annotation values in classes annotated to use `Checker` as their Proxy must be changed with the new `AccountChecker` name. Using the connector aspect changes must also be applied, but they are all localised in the same file (i.e. the connector aspect).

Chapter 4

Specialised aspects for aspect-oriented design patterns

The design patterns' implementations discussed in this chapter are based on the ones already discussed in chapter 3. For each proposed pattern's implementation (i.e. an aspect) a correspondent template version is derived, used to generate a *Specialised Aspect* (SA). The already described reflective AODPs, albeit bringing an enhanced modularity, might be slower than their respective object-oriented implementations, mainly for the extensive use of introspective instructions and dynamic invocations. The SA variants described in this chapter are put forward as a faster alternative, also easier to read, of their respective AA and CA versions. Such versions need not use the already defined annotations.

Such aspects are derived from a *template* that does not mention any specific class in its code, instead it mainly uses the roles' name (in place of the ones of the involved classes) that should be played in the DP. A template offers a set of code fragments –mainly being pointcuts, advices, and member variables– related to abstract roles' names. A template's purpose is just to be a model, thus it is general, yet not intended to be used directly. It defines the essential behaviour of a design pattern without being coupled with any specific class.

Given an aspect template and a role–class mapping for a design pattern, such templates are used to automatically generate an aspect to be woven into a specific application to enforce the roles (and behaviour) of the pattern for that application. The aspect is created from template fragments by mapping the roles' names to actual classes of an application, thus being a completely working and usable aspect for the specific application it is generated to. A SA is created for each instance of a design pattern.

There are two fundamental differences between the AA versions already discussed and the SA ones of this chapter. The generated, specific, aspect has no need to use the reflective API to access information at runtime, thus it avoids the overhead introduced with the AA versions of the design pattern. Hence, such specialised aspects need not use a caching mechanism.

The approach using these specialised aspects need not use annotations marking classes, as both pointcuts and advices are generated with the specific involved classes for each role.

All the properties already discussed in section 1.1 still hold true for these versions. In particular, the aspects that will be shown in the sequel, as the ones already discussed, completely fulfil the *SPoC* property. In fact, given an application and any version of an aspect (i.e. AA, CA or SA) for a design pattern, the behaviour of the pattern can be added (or removed) from the application just by weaving an aspect. This is exactly how the performance measures (section 5.2) were performed for all the versions of a design pattern.

For example, using the Eclipse IDE [Pro11], the execution of the code of an application using the `SingletonPattern` or the `SingletonPatternSA` (described in the following), is just a matter of mutually excluding one of the two from the build path, compiling and running the application. The application will behave in the same way in both cases, however in the former case the aspect using reflection is executed while in the latter it is the specialised aspect that is being executed. No changes to any other class are needed.

The aspects' generation is mainly intended to be used to produce the design patterns' code when developing a new application. However, the specialised aspects can also be used in legacy object-oriented applications already implementing a pattern: such applications can be converted to use the aspect-oriented implementations for the related pattern. For this case some refactoring steps have to be performed to alter the legacy classes to be able to use the SA for an AODP. For example, when the `SingletonPattern` (in any version) is used to convert a `SingletonBank` class into a `Bank` class that uses the `SingletonPattern` to play the Singleton role, there might be some classes, clients of `Bank`, whose code uses the `getInstance()` of `SingletonBank` and have to be modified to use the `new` instruction as the new

```
public aspect SingletonPatternSA {
    Singleton obj = null;

    pointcut trapCreation(): call(Singleton.new (..));

    Singleton around() : trapCreation() {
        if (obj == null) {
            obj = proceed();
        }
        return obj;
    }
}
```

Figure 4.1: *The* SingletonPatternSA *aspect's template*

`Bank` class allows.

In the next sections the solutions for the proposed implementations of SAs are described, mainly highlighting the differences with their respective AA counterparts. In section 4.6 are described the generation phases of the aspects and the refactorings for converting legacy classes.

4.1 `SingletonPatternSA` aspect

The template aspect for the *Singleton* DP is shown in figure 4.1. The intent (and behaviour) of the generated aspect remains the same as the previous version (thus inheriting both *SoC* and *SPoC*), that is to capture the invocations of `new` on classes defined as playing the Singleton role (say `Bank`), so to let the clients' classes to invoke `new` instead of the (static) `getInstance()` method. Thus any client can invoke the following

```
Bank b = new Bank();
```

to obtain a reference to the only instance of `Bank`, either freshly created by the `SingletonRolesBank` aspect (figure 4.2) or already created before (thus verifying *REoR*).

A sample generated aspect is shown in figure 4.2, with which a `Bank` class is made a Singleton. At a first glance, the template may seem very similar to the original

```
1  public aspect SingletonPatternBank {
2      Bank obj = null;
3
4      pointcut trapCreation(): call(Bank.new(..));
5
6      BankAsp around() : trapCreation() {
7              if (obj == null) {
8                  obj = proceed();
9              }
10             return obj;
11     }
12 }
```

Figure 4.2: *A sample specialised aspect from the* SingletonPatternSA *template*

placeholder	value
SingletonPatternSA	`SingletonPatternBank`
Singleton	`Bank`
SingletonPackage	`BankApp`

Table 4.1: *Sample substitutions for the* SingletonPatternSA *generation*

AA aspect (figure 3.2). Although they may look similar, there are fundamental differences between them.

The first difference between the AA version and this one is that the `SingletonPatternBank` aspect makes no reference to the `@Singleton` annotation, i.e. for the developer there is no need to explicitly mark the Singleton (`Bank`) class to make it play such a role, once the aspect is generated, all the programmer has to do is to weave the aspect into the application, the `SingletonPatternBank` will intercept only `new` invocations on the `Bank` class, as specified by the pointcut in line 4.

The advice code is basically the same as in the AA version, as they share the same black-box behaviour, but no reflective calls are made: the advice code is aware that it manipulates `Bank` instances. Thus, as it will store the only instance of `Bank`, the `singles` map used in the AA version (line 2, figure 3.2) is not needed anymore.

The specific aspect can be automatically generated using values such as the ones shown in table 4.1, i.e. to make a `Bank` class a Singleton. The generated aspect is meant to be used only for the application it is tailored to, and contains in a single

```
public aspect FlyweightPatternSA {
private Map<Integer, ConcreteFlyweight> flies =
            new Hashtable<Integer, ConcreteFlyweight>();

pointcut trapCreation(Object k): call(ConcreteFlyweight.new(..)) && args(k);

  ConcreteFlyweight around(Object k) : trapCreation(k) {
    ConcreteFlyweight ref = null;
    Integer hash = MyHashing.getHash(k);
    ref = flies.get(hash);
    if (ref == null) {
      ref = proceed(k);
      flies.put(hash, ref);
    }
    return ref;
  }
}
```

Figure 4.3: *The* FlyweightPatternSA *aspect's template*

module all the code needed for the specific class to implement a Singleton behaviour, thus the *ECoR* property holds true.

4.2 `FlyweightPatternSA` aspect

The template aspect encapsulating the *Flyweight* DP behaviour is shown in figure 4.3. All the code needed to implement the behaviour of the DP is put in the aspect, thus *ECoR* is satisfied.

The class playing as a ConcreteFlyweight is left untouched when using the aspect, and thus can implement just its domain code and being reusable (*SoC*); no FlyweightFactory is needed as its responsibility is carried out by the aspect.

The aspect intercepts the creation of classes playing the ConcreteFlyweight role providing the caller with an instance identified by the key parameter. A client needing a ConcreteFlyweight instance would simply invoke `new`, without having to resort to a FlyweightFactory as it happens in the regular object-oriented solution. Thanks to this enforcing, as in the AA and CA versions, the *REoR* property is satisfied. The management of the keys used to retrieve a specific instance of a

```
1  public aspect FlyweightPatternMyCharacter {
2    private Map<Integer, MyCharacter> flies=new Hashtable<Integer, MyCharacter>();
3
4    pointcut trapCreation(Object k): call(MyCharacter.new(..)) && args(k);
5
6     MyCharacter around(Object k): trapCreation(k){
7         MyCharacter ref = null;
8         Integer hash = MyHashing.getHash(k);
9         ref = flies.get(hash);
10        if (ref == null) {
11            ref = proceed(k);
12            flies.put(hash, ref);
13        }
14        return ref;
15    }
16 }
```

Figure 4.4: *A sample specialised aspect from the* FlyeightPatternSA *template*

placeholder	value
ConcreteFlyweight	`MyCharacter`
FlyweightPatternSA	`FlyweightPatternMyCharacter`

Table 4.2: *Sample substitutions for the* FlyweightPatternSA *generation*

ConcreteFlyweight is the same as in the AA version, by the same `MyHashing` class.

The template holds no references to the `@Flyweight` annotation nor to the reflective calls used in the AA `FlyweightPattern`, however its observable behaviour is the same.

If a `MyCharacter` class has to behave as a ConcreteFlyweight, the substitutions needed for the generation of the specialised aspect (figure 4.4) are, e.g., the one shown in table 4.2.

The `FlyweightPatternMyCharacter` aspect intercepts all `new` invocations on the `MyCharacter` class from any client class. Once a `new` is trapped its argument is hashed (line 8) so to properly recognise which instance of `MyCharacter`, stored in the `flies` map, to return to the caller. So a client class can directly create a `MyCharacter` object and, based on the arguments, will receive a reference without having to be coupled with a FlyweightFactory (which does not exist in this version of the design pattern).

A difference with the AA version is apparent comparing the definition of the `flies` map on the two aspects. The AA version, being general, needs to tell apart different classes playing the `ConcreteFlyweight` role, and for each class it stores a reference for each possible (hashed) argument. In the SA version this is not needed. The SA version is customised for a specific class (such as `MyCharacter`) and the `flies` map stores just the references to the instantiated `MyCharacter` objects, indexed by the (hashed) argument used for the creation. If another class (say `Weather`), even in the same application, has to be treated as a ConcreteFlyweight, it is sufficient to generate another aspect (say `FlyweightPatternWeather`), as it will hold its own `flies` map, with `Weather` objects as values.

Removing the aspect from the application makes the `MyCharacter` class a regular class, i.e. not a ConcreteFlyweight anymore, hence making it reusable in any other context as it is, i.e. without changes. This accommodates for the *SPoC* property.

4.3 `ProxyPatternSA` aspect

The template for the `ProxyPatternSA` aspect is shown in figure 4.5. As with other specialised aspects, it just holds the references to the roles to be played instead of real classes. All the code implementing the *Proxy* behaviour for a specific set of classes is obtained by generating the specific aspect from the template version. A single aspect holds all the behavior of an instance of a *Proxy* design pattern, thus ensuring both *SoC* and *ECoR*, as the aspect implements all the behaviour for the pattern in a single module without implementing other concerns.

All the clients of a `RealSubject` (intended both as a role and as a class), directly access it while the aspect will enforce the routing to the `Proxy`. So a client (and a programmer) can not mistakenly access a `RealSubject`, thus satisfying *REoR*. Thus, when a `RealSubject` class is not required to play this role for evolution purposes, both clients and `RealSubject` remain the same. Also to remove the shielding of the `RealSubject` class it suffices to remove the `ProxyPatternSA` from the application (ensuring *SPoC*).

The `ProxyPatternSA` aspect makes no reference to the `@Proxy` annotation. No reflective call is used in its code, so, e.g., the `trapCreation` advice can directly perform a `new` invocation on a `Proxy` class, while the AA version, to be general,

```
public aspect ProxyPatternSA {
   private Map<RealSubject, Proxy> proxies =
               new WeakHashMap<RealSubject, Proxy>();
   private RealSubject tmp = null;

   pointcut trapCreation(): call(RealSubject.new(..));

   Object around() : trapCreation() && !within(Proxy) {
      tmp = (RealSubject) proceed();
      Proxy newp = new Proxy();
      proxies.put(tmp, newp);
      return tmp;
   }

   Object around() : trapCreation() && within(Proxy) {
      return tmp;
   }

   pointcut omit(RealSubject o) :
         target(o) && !within(Proxy);

   Object around(RealSubject o): trapCalls(* RealSubject.request1(..)) && omit(o){
      return proxies.get(o).request1();
   }
}
```

Figure 4.5: *The* ProxyPatternSA *aspect's template*

has to perform an equivalent instantiation calling the `newInstance()` method and reading the value of the `@Proxy` annotation.

To generate a usable aspect to implement a specific instance of the *Proxy* design pattern a possible mapping for some placeholders values are shown in table 4.3. An abridged version of the complete generated aspect to force the `Point` class to play the RealSubject role is shown in figures 4.6 and 4.7, the aspect is meant to be used just for the pair `Point` and `ProxyPoint`.

The differences with the AA version seem more discernible in this case, which, by the way, is more paradigmatic with respect to other specialised aspects of design patterns presented in this chapter.

placeholder	value
RealSubject	`Point`
Proxy	`ProxyPoint`
Subject	`PointInterface`
request1	`setX`
request2	`getY`

Table 4.3: *Sample substitutions for the* ProxyPatternSA *generation*

The `proxies` map aims at keeping the pairs (RealSubject, Proxy), however there is no need to have both the key and the value defined as generic `Object` type, as the aspect is generated to hold, respectively, a `Point` and a `ProxyPoint` (cf. table 4.3). Similarly, the `tmp` variable is declared to be a `Point`, instead of an `Object`.

The `trapCreation` pointcut (line 6, figure 4.6) is combined with other pointcut designators to distinguish three[1] possible cases. I.e. the single (general) advice needed for the AA version, is split in three different advices, all capturing different executions of `new` for a specific RealSubject (a `Point`). The generated code refers to a `Point` class with two constructors, one without parameters and one with an `int` parameter.

The first advices (lines 8 and 15, figure 4.6) hold essentially the same code, however they accommodate two possible invocation of `new` on the `Point` class: with (line 15) and without a parameter (line 8). This is necessary as the aspect does not use reflection and have to treat each case in a different advice. Please remember that albeit this code is duplicated, it is automatically generated.

These advices capture the instatiation of a `Point` class and put its reference in the `proxies` list, pairing it with the Proxy (a `ProxyPoint`) automatically assigned, yielding the exact behaviour obtained when alternatively using the AA version.

The `trapCreation` pointcut-related advice of the AA version, in its code, distinguishes between different `new` invocations. It recognises the ones made by a Proxy and, instead of allowing the instantiation, returns its paired RealSubject reference from the `proxies` map. This is now done via the advice in line 22 (figure 4.6), which is executed when a `new` invocation made by a `ProxyPoint` (i.e. a Proxy) is intercepted: i.e. the code is now split in different advices instead of being in the

[1]The exact number is dependent on the application, as it reflects the different constructors existing for a RealSubject class. For more details on the generation please see section 4.6.

```
public aspect ProxyPatternPoint{

private Map<Point, ProxyPoint> proxies = new WeakHashMap<Point, ProxyPoint>();
private Point tmp = null;

pointcut trapCreation() : call(Point.new (..));

Point around() : trapCreation() && !within(ProxyPoint) {
    tmp = (Point) proceed();
    ProxyPoint newp = new ProxyPoint();
    proxies.put(tmp, newp);
    return tmp;
}

Point around(int p0) : trapCreation() && args(p0) && !within(ProxyPoint) {
    tmp = (Point) proceed(p0);
    ProxyPoint newp = new ProxyPoint(p0);
    proxies.put(tmp, newp);
    return tmp;
}

Point around() : trapCreation() && within(ProxyPoint) {
    return tmp;
}
...
```

Figure 4.6: *A sample specialised aspect from the* ProxyPatternSA *template (part 1 of 2)*

same one and dependent on a conditional statement. These advices handle all the possible cases for the creation of a RealSubject.

The `trapCalls` advice is substituted with all of the pointcuts and advices in figure 4.7: such pointcuts and advices implement the same behaviour of the said advices of the AA version. Any method call on a `Point` object is routed to a method with the same name of the `ProxyPoint` associated object. The generated code defines a set of pointcuts and related advices, instead of only one, to perform this kind of invocations. Every public method of the original `Point` class are used to generate pointcuts such as the ones in figure 4.7. The pointcuts and advices shown in figure 4.7 are just a meaningful subset of the generated ones.

```
1  pointcut omit(Point o) : target(o) && !within(ProxyPoint);
2
3  void around(Point o, int p0) : call(* Point.setX(int)) && args(p0) && omit(o) {
4      proxies.get(o).setX(p0);
5  }
6  void around(Point o) : call(* Point.resetY()) && omit(o) {
7      proxies.get(o).resetY();
8  }
9
10 pointcut omit1() : !within(ProxyPoint);
11
12 void around(Point p0) : call(* Point.setStaticPoint(Point)) && args(p0) && omit1() {
13     ProxyPoint.setStaticPoint(p0);
14 }
```

Figure 4.7: *A sample specialised aspect from the* ProxyPatternSA *template (part 2 of 2)*

Additional poincuts such as `omit` and `omit1` are generated as reusable pointcuts, e.g. the `omit` pointcut is used for two advices (line 3 and 6) to filter calls, as calls to `Point` taking place inside the `ProxyPoint` class are allowed as it is a Proxy.

When the `omit` pointcut is used in conjunction with the pointcut in line 3, it supports the advice in identifying calls to the `Point.setX(int)` method, thus letting the advice to execute the `setX()` method, with the same captured actual parameters, on the `ProxyPoint` object associated with the `o` reference in the `proxies` map. When the `omit` pointcut is used with the advice on line 6, it supports the advice to capture the `resetY()` method, without parameters. The `omit1` pointcut is used for invocations where the reference to the target object is not needed, such as for a static method invocation. The advice in line 12 is an example of such an advice. It is executed when the `Point.setStaticPoint()` is invoked, and instead of letting its execution go on, it calls the method with the same name but on the Proxy.

4.4 `ObserverPatternSA` aspect

The `ObserverPatternSA` aspect template (figure 4.8) contains a pointcut, `trap-Observed` (lines 5–6), to intercept the observed methods of a ConcreteSubject (e.g. named `observedMethod1()`, as in figure), i.e. the methods that change the observable

```
1  public aspect ObserverPatternSA {
2    private Map<ConcreteSubject, List<Observer>> subjects =
3          new WeakHashMap<ConcreteSubject, List<Observer>>();
4
5    pointcut trapObserved(ConcreteSubject obj) :
6          this(obj) && execution(* ConcreteSubject.observedMethod1(..));
7
8    after(ConcreteSubject obj) : trapObserved(obj) {
9      List<Observer> observers = subjects.get(obj);
10     if (observers != null)
11       for (int i = 0; i < observers.size(); i++) {
12         observers.get(i).update(obj);
13       }
14   }
15
16   public void addObserver(ConcreteSubject subj, Observer obs) {
17     List<Observer> observers = subjects.get(subj);
18     if (observers == null)
19       observers = new LinkedList<Observer>();
20     observers.add(obs);
21     subjects.put(subj, observers);
22   }
23 }
```

Figure 4.8: *The* ObserverPatternSA *aspect's template*

state of the ConcreteSubject, and a paired advice (lines 8–14) that is triggered after the execution of such a method and informs all the registered ConcreteObservers by calling their `update()` method with the captured ConcreteSubject reference (this satisfies both *SoC* and *ECoR*).

This behaviour satisfies the *REoR* property as the programmer has no need to explicitly call the `update()` method in its code, as it is the woven aspect that will enforce such calls. Please note how this specialised aspect is free of both the annotation's and reflection API use.

The aspect keeps the `subject` map to associate each ConcreteSubject with the list of its ConcreteObservers. The key for the map is a specific `ConcreteSubject` class, and so the values of the list of `ConcreteObservers`. The list is managed by the `addObserver()` method (lines 16–22), which does exactly the same as the AA

```
1  public aspect ObserverPatternMyData {
2      private WeakHashMap<MyData, List<DataObserverInterface>> subjects =
3          new WeakHashMap<MyData, List<DataObserverInterface>>();
4
5      pointcut trapObserved(MyData obj):
6          this(obj) && execution(void MyData.addMember()) ;
7
8      after(MyData obj): trapObserved(obj) {
9          List<DataObserverInterface> observers = subjects.get(obj);
10         if (observers != null)
11             for (int i = 0; i < observers.size (); i++) {
12                 observers.get(i).update(obj);
13             }
14     }
15
16     public void addObserver(MyData subj, DataObserverInterface obs) {
17         List<DataObserverInterface> observers = subjects.get(subj);
18         if (observers == null)
19             observers = new LinkedList<DataObserverInterface>();
20         observers.add(obs);
21         subjects.put(subj, observers);
22     }
23 }
```

Figure 4.9: *A sample specialised aspect from the* ObserverPatternSA *template*

version, but aware of holding specific classes' references, instead of generic objects.

To generate the fragment of the specific aspect shown in figure 4.9 the needed substitution are shown in table 4.4. The `ObserverPatternMyData` automatically notify all the registered ConcreteObservers stored in the `subjects` map when the `addMember()` method changes the state of any object of the `MyData` class (which is identified with its own list of observers, by its reference, i.e. the `obj` reference captured in line 6 by the pointcut). The generated aspect can be woven or removed at will without changes in both client and roles classes, thus ensuring *SPoC*.

placeholder	value
ConcreteSubject	`MyData`
Observer	`MyDataObserverInterface`
observedMethod1	`addMember`

Table 4.4: *Sample substitutions for the* ObserverPatternSA *generation*

placeholder	value
Component	`Resource`
Composite	`Dir`
operation1	`show`

Table 4.5: *Sample substitutions for the* CompositePatternSA *generation*

4.5 `CompositePatternSA` aspect

The template aspect (figure 4.10) implements all the operations needed to manage the lists (the `comps` map) of Component objects (so either Leafs or Composites), allowing the `Composite` class to be free from implementing such code (fulfilling *SoC* and *ECoR*).

The `CompositePatternSA` aspect intercepts all the calls to a `operation1()` method on a Component object and automatically calls the same method on all its children. As in the AA version, after the `operation1()` method has been called on a child, it is also called the same method, with an additional parameter, on the Composite object so to allow the collection of intermediate results obtained by the children.

The template is used to generate specific aspects, such as the (fragment of) one shown in figure 4.11. Some of the substitutions needed to generate the `Composite-PatternFileSystem` are shown in table 4.5. The `addChild()` method is the only extra method shown.

The `comps` map stores `Resources`, the interface from which a `Dir` class (i.e. a Composite) inherits. The `trapOperations1` pointcut intercepts all the calls to the `show()` method advice on a `Dir` object. Similar advices are generated for any other `operation()` method of the `Dir` class, these are not shown in figure. Once a `show()` method is intercepted by the aspect it will be executed on each children of the `co` object, alternating it with passing the return value (an `int`) to the `show(int)`

```
public aspect CompositePatternSA {
    private Map<Component, List<Component>> comps =
                            new Hashtable<Component, List<Component>>();

    pointcut trapOperations(Composite co) :
        call(int Component.operation1(..)) && target(co);

    Object around(Composite co) : trapOperations(co) {
        List<Component> children = comps.get(co);
        if (children != null)
            for (int i=0; i<children.size(); i++)
                co.operation1(children.get(i).operation1());
        return proceed(co);
    }

    public void addChild(Component comp, Component child) {
        List<Component> childLst = comps.get(comp);
        if (childLst == null) childLst = new LinkedList<Component>();
        childLst.add(child);
        if (childLst.size() == 1) comps.put(comp, childLst);
    }

    public List<Component> getChildrenList(Object comp) {
        return comps.get(comp);
    }
}
```

Figure 4.10: *The* CompositePatternSA *aspect's template*

method of `co`. After all the children are visited by the `show()` method, this is called on the original `co` object it was trapped to. Please note that this aspect does not need to use the `trapRecursiveOperations` pointcut (and related advice), to allow the recursion on the children of the children of a `Dir`.

To remove the Composite role from the `Dir` class it suffices to remove the aspect from the application, thus *SPoC* is verified. The *REoR* property is also verified as the programmer has no need to write the code for the handling of the children's list.

```
public aspect CompositePatternFileSystem {
   private Map<Resource, List<Resource>> comps =
                  new Hashtable<Resource, List<Resource>>();

   pointcut trapOperations1(Dir co) :
      call(int Resource.show(..)) && target(co);

   int around(Dir co) : trapOperations1(co) {
      List<Resource> children = comps.get(co);
      if (children != null) {
         int res;
         for (int i = 0; i < children.size(); i++) {
            res=children.get(i).show();
            co.show(res);
         }
      }
      return proceed(co);
   }

   public void addChild(Resource comp, Resource child) {
      List<Resource> childLst = comps.get(comp);
      if (childLst == null) childLst = new LinkedList<Resource>();
      childLst.add(child);
      if (childLst.size() == 1) comps.put(comp, childLst);
   }
}
```

Figure 4.11: *A sample specialised aspect from the* CompositePatternSA *template*

4.6 Generation of specialised aspects

The presented aspects' templates are used to create concrete aspects to be woven into real applications. The generation of the aspects described in the previous sections is intended to be used when the programmer develops an application aware of the AODPs since the start of the development. This means that client classes are developed so that they can use the AODPs version, e.g. a client for a Singleton will not use a `getInstance()` method but will use `new`, as the Singleton role will be automatically superimposed by weaving the `SingletonPatternSA` aspect with the application.

In this case the programmer has to generate an aspect for each instance of the design patterns needed by the application, so if both classes `Bank` and `Account` have to play the `Singleton` role, two specialised aspects (namely, `SingletonPatternBank` and `SingletonPatternAccount`) have to be generated.

To perform the creation it is mandatory to identify the classes of the application involved in the design pattern's role superimposition, i.e. to define a mapping between roles played in the design pattern and classes that should play these roles. This can be done both manually, when the programmer knows which classes play which role in the application, or automatically, using one of the available approaches found in literature [PT06, DZS09, TCSH06].

The simplest case of SA generation is the substitution of role names in the template aspect with class names to play that role, as in the tables already shown (such as tables 4.4 and 4.3).

The generation of the SAs is performed in, basically, the same fashion for all of the presented design patterns, with minimal differences among them. The case of the *Proxy* design pattern will be analysed here since it encompasses the techniques used for other SA generations.

The template is composed by some variable parts, such as the class placeholders' names, and fixed parts, such as lines 11–12 in figure 4.5. Moreover, some pointcuts and advices can be repeated in the final SA, because they can be applied to more than one method call. For example, the `trapCreation` advice (lines 8–13 in figure 4.5) has to be used for both possible constructor calls in the sample application, once for the `Point()` constructor (which yields the advice in lines 8–13 figure 4.6) and once for the `Point(int)` constructor (which yields the advice in lines 15–20). Please note that not every part of the aspect might need to be repeated, as the `proxies` map definition and the `tmp` declaration (lines 2–4 in figure 4.5) are unique for each pattern instance. The information on which parts of the template should be repeated, at this stage, are hardcoded in the generation tool (called *SpecialiseAspect* or `sa`).

Essentially, the `sa` tool internally stores the design pattern's template as strings, each string has to be repeated and/or modified to adapt the aspect to the mapping used for a specific application. The classes specified in the mapping have to be reflectively analysed to (i) verify the existence of classes and methods defined in the mapping, and (ii) to obtain the signature for the involved methods (i.e., return type and parameters list).

For the *Proxy* design pattern the classes that have to play the RealSubject and Proxy roles are (reflectively) checked to make sure they implement the same Subject interface, as mandatory for the pattern itself[2], should this control fail the generation is aborted.

The `sa` tool repeats its `addTrapCreation()` and `addTrapCall()` methods respectively one time for each constructor and method found in the Subject interface, each time substituting the role name with the name of the class that has to play it. The `proxies` map definition and `tmp` variable are inserted just once. The name of the aspect is uniquely generated as a combination of both the pattern name and the involved class names, so to avoid conflict with other specialised aspects involving other classes.

The output of `sa` is written as a file, to be used with the application. A sample of a specialised aspect has been shown in figure 4.5, generated with the mapping shown in table 4.3. The generated aspect will be compiled using the `ajc` weaver[3].

4.6.1 Refactoring of existing applications

The generation of specialised aspects is a mandatory phase when dealing with existing object-oriented applications and when the superimposition of roles to classes is desired. Beside the specialised aspects' generation, some additional refactoring steps have to be performed on the existing classes that already use object-oriented design patterns to let them use an SA version of the pattern. For example a client class accessing a Singleton has to be changed not to use the `getInstance()` method[4].

[2]Similar controls are performed for other design patterns, e.g. in the Singleton case the tool checks that the designated class has no public constructor.

[3]Developers could change the code of classes and/or aspects for further evolution purposes, then the `ajc` weaver will automatically perform checks on classes and aspects that assess whether expected classes, methods and parameters are actually implemented. Checks at weaving-time are performed for methods invoked and declared variables, however no alert is given when some pointcuts do not match any point on the code. This is known as *fragile pointcuts problem* [Lad09]. In the proposed apporach, the generation of aspects according to the designated roles for existing classes makes the fragile pointcuts problem less significant, i.e. when the interface of classes are changed it will suffice to generate specific aspects again and these will accordingly match the new interfaces.

[4]Several researchers investigated refactorings of object-oriented applications to convert them into equivalent aspect ones, a notable example being [HMK05].

Some of the changes needed have been sketched in the AODPs description, in the following sections they will be described in greater detail.

Proxy refactoring

The structure of the *Proxy* design pattern is so that the removal of a Proxy from an existing object-oriented application is facilitated, as only the client classes of the Proxy are coupled with the Proxy class. The class acting as a RealSubject is not coupled with its Proxy, so it need not changes, while a Proxy class is allowed to access to its RealSubject, so its call must be preserved. These classes are preserved from the refactoring[5].

A client class may instantiate, or use, a Proxy in two ways: using a Subject member or a Proxy one. When a `p` variable is instantiated, in both cases, a Proxy object has to be instantiated. So if the programmer wants to use the AODP version of the *Proxy* design pattern, once the `sa` tool generates the SA, it suffices to examine the object-oriented application to find out all the references to the Proxy class in non-role classes, given in the class–role mapping.

The refactoring steps are repeated for each reference of a Proxy (or Subject) class found in non-role classes. Any Proxy variable is changed with the respective RealSubject one, found in the mapping; if necessary, the type of the variable is also changed into the Subject interface. E.g. for the mapping in table 4.3, the following instruction

```
ProxyPoint p = new ProxyPoint(3,7);
```

is changed in

```
PointInterface p = new Point(3,7);
```

as the `ProxyPatternPoint` aspect will automatically take care of the `ProxyPoint` instantiation by intercepting the `Point` creation.

For a `p` reference, both a Proxy or a Subject, its declaring type is changed, all the method calls on that variable remain valid. This holds true by the definition of a Proxy, as both Proxy and RealSubject have to implement the same Subject interface and so the same public methods. Thus any existing method call, say `p.show()`,

[5] In the following, the classes which do not play any specific role for a design pattern are simply called non-role classes.

made on a Proxy object remains valid when performed on a RealSubject object, and the original code need not to be changed.

After the refactoring has been performed the resulting client classes are no more coupled with the Proxy class, the *Proxy* behaviour is obtained by weaving the SA with the application.

Singleton and ***Flyweight*** refactoring

Both *Singleton* and *Flyweight* AODPs share similar refactoring steps, as both are invoked by their clients only to obtain a reference to an object. In the *Singleton* case a client invokes the static `getInstance()` method on the Singleton class, while in the *Flyweight* one a client invokes a FlyweightFactory's method passing a key as an argument.

In the Singleton class, found in the class mapping, all the private constructors have to be found and changed to become public. All the methods in the Singleton class returning a Singleton object are checked to identify the `getInstance()` one, once found this method can be removed from the class (its name is internally stored for the following step). As the `getInstance()` method could perform arbitrary operations before the instantiation of the Singleton object (e.g. memory constraints checks), only the simple and general case shown in [GHJV94] is considered. The static private member variable of Singleton class is removed[6], as it will be stored in the generated SA. The specific name found for the `getInstance()` method is discovered in all the non-role classes of the application. Once found it has to be changed to the equivalent call to the (now public) constructor of the Singleton class.

The object-oriented *Flyweight* removal is very similar. All client classes invoking a FlyweightFactory method returning a ConcreteFlyweight reference have to be changed to use the regular constructor. An invocation such as

```
MyCharacterInterface c = CharacterFlyweightFactory.getChar(x);
```

is converted to

```
MyCharacterInterface c = new MyCharacter(x);
```

using table 4.2 as a possible mapping. The FlyweightFactory class can be removed from the application.

[6]Supposing only one of such variable is declared.

Observer refactoring

To remove the object-oriented version of the *Observer* design pattern the following steps have to be performed. Given the class mapping, the class playing as a ConcreteSubject has to be changed to not extend the Subject superclass, so effectively keeping it free to extends other domain classes instead of the pattern-related one. This is done by removing the `extends Subject` in the class definition.

Since the object-oriented pattern relied on its (former) superclass it used the `notify()` and `setState()` methods provided by the Subject superclass. Calls to these methods are searched in the body of all methods of the ConcreteSubject class and removed, as they are implemented by the generated `ObserverPatternSA`.

In the `ConcreteObserver` classes no changes need to be done, as their `update()` method will be called by the generated specialised aspect.

All the non-role classes in the application must be checked to find out any call to the `attach()` or `detach()` methods. These calls must be changed to use the equivalent alternatives provided by the specialised aspect. So, given the mapping in table 4.4, and given the `a` and `v` objects defined as in the following

```
MyDataObserver v = new MyDataObserver(); // ConcreteObserver
MyData a = new MyData(); // ConcreteSubject
```

the following instruction

```
a.attach(v);
```

found in a non-role class, is changed to

```
ObserverPatternMyData.aspectOf().addObserver(a, v);
```

thus obtaining the same effect of the original instruction.

Composite refactoring

To change an object-oriented *Composite* design pattern all the utility methods (`add()`, `remove()` and `getChildren()`) must be changed from calls to the inherited Component methods to the methods provided by the SA. For example, using the mapping in table 4.5, if a `dir` object plays the Composite role, the following

```
dir.add(f);
```

is changed to

```
CompositePatternFileSystem.aspectOf().addChild(dir, f);
```

As the order of the parameters of the object-oriented `add()` method is different from the SA version's, some changes in their order have to be performed. I.e. the `dir` object has to be passed as the additional argument to the `addChild()` method.

From each Composite class (found by the mapping) the member variable storing its children list has to be removed, as such lists are stored in the specialised aspect. To find out the involved variable all the aforementioned utility methods are analysed, as they all are supposed to access and manage exactly this list. The list can be found by enumerating the objects modified by the `add()` and `remove()` methods (just one, in the simplest case), comparing with the object used in a loop in the `getChildren()` method, if all three methods use the same reference a match is found and the related variable is assumed to be removable. In case of multiple matches the programmer is asked to provide which variable to remove.

The last changes to be performed are related to the `operation()` methods listed in the mapping. Such methods loop on all the children of a Composite and perform specific operations on them. All accesses to the member variable holding the children list are removed, while the relevant code performing the operation on each child should remain; the result (if any) computed by this code, will appear as an input parameter of the amended `operation()` method to be implemented by the Composite to collect the results on its children.

Chapter 5

Assessment of aspect-oriented design patterns

The previous chapters put the focus on the mechanisms that allow the proposed AODPs to perform as a reasonable alternative to the original object-oriented ones. In this chapter, section 5.1 illustrates the benefits that can be obtained by the adoption of AODPs. They are evaluated in terms of advantages for the developer and the resulting application modularity with respect to the object-oriented counterparts, showing how known problems[1] of classical object-oriented implementations can be avoided using the proposed AODPs. Section 5.2 deals with the performance assessment of the proposed AODPs. A sample application using the proposed implementations has been developed for both approaches, the regular object-oriented and the aspect-oriented one. Both applications are functionally equivalent, the only difference is how the design patterns are implemented. The AODPs have been implemented in all the versions available for a given pattern, i.e. all AA, CA and SA versions have been implemented where possible.

5.1 Overall assessment

A developer can choose to use both categories of proposed AODPs, the reflective version (AA or CA versions) or the generated, non-reflective alternative (SA version). Both alternatives provide similar advantages over the standard object-oriented implementation of a design pattern.

[1] Several authors identify such problems, e.g. [HK02, HB02, Bos98].

When a developer implements a design pattern using object orientation the code of the involved classes is not compactly defined in a single module, instead it ends up scattered over different classes of the application. This also means that the involved classes become more complex to manage, to understand and to reuse. Moreover, as the code implementing the pattern is dispersed over the application classes, if a developer is unaware of such an implementation of the pattern, the presence of the latter in the system is not evident. On top of that, the classes implementing roles for a design pattern are not the only concerned by reusability and modularity problems. Also the client classes of the role-implementing ones may have to be especially tailored to fit the pattern format when the latter is accessed.

In the object-oriented *Observer*, for example, the ConcreteSubject class has to be modified to host a list of ConcreteObservers and the code for the management, e.g. the `attach()` or `notify()` methods. Such code could be provided by inheritance, object composition or directly added to the ConcreteSubject class. However, such methods are not related to the main domain responsibilities of the ConcreteSubject, they are provided only to superimpose the behaviour for the role the class has to play. Moreover, selected methods of the ConcreteSubject have to contain calls to the `notify()` method, so to properly update the ConcreteObservers. Thus such methods contain both domain code (such as the domain code which updates the internal state of the ConcreteSubject) and pattern-related code (to notify the ConcreteObservers of the state change). A ConcreteSubject class should, if possible, be reused, however this additional non-domain responsibility makes it coupled with pattern-related code and thus makes it less reusable. The ConcreteSubject class also becomes more difficult to understand.

By using one of the `ObserverPattern` aspects proposed, a ConcreteSubject class is not complicated with the list of ConcreteObservers or pattern-related methods as these are defined in the aspect, not in the role-implementing class. The ConcreteSubject's methods can just be annotated using a connector aspect (section 3.7) to define methods which the ConcreteObservers are interested in. This implies that the `notify()` call is not mixed with the ConcreteSubject code, as such call is automatically activated by pointcuts defined in the aspect by means of the said annotations. The resulting ConcreteSubject class is thus simpler, contains only its domain-related code and is more reusable. By using the the generated specialised aspect for the *Observer* design pattern similar results are obtained. The ConcreteSubject class does not have to mix its code with the pattern-related one, as a generated aspect contains the hooks and code for the specific classes to let the `notify()` method be automatically invoked after an observed method is executed.

The object-oriented *Observer* implementation is *tangled* with the class implementing the ConcreteSubject role, i.e. the code implementing it is interspersed with the domain code of the ConcreteSubject class. This makes the pattern implementation difficult to recognise for the developer, as a class has to be analysed with care to understand if the pattern is implemented and by what methods.

In contrast to the object-oriented implementation, using the AODP *Observer*, the code for the pattern is all defined in a single module (the `ObserverPattern` aspect) and by reading the annotations contained in the connector aspect the programmer can immediately recognise the classes involved for both roles (ConcreteObserver and ConcreteSubject). Instead, to recognise the occurrence of the design pattern in an object-oriented implementation the programmer is forced to first recognise what class plays the ConcreteSubject role, then to identify, e.g., the `attach()` method (it might be called differently) and finally search all the classes of the application for calls to the said `attach()` method so to understand the involved ConcreteObservers classes. The same advantage over the object-oriented version is also available in the proposed SA version. An occurrence of the pattern is encapsulated in a single aspect, which, when visually inspected, makes clear for the programmer what classes are involved in the design pattern.

Another problem typical of the object-oriented implementation of many design patterns is the coupling of the application classes with the role-implementing ones. This can be shown using the *Proxy* design pattern as an example. In an object-oriented implementation a client class accessing a RealSubject is tightly coupled with its shielding Proxy class. As the client should not directly instantiate a RealSubject but can, and should, instantiate a Proxy instead. This solution hinders the reusability and evolution of such client classes. Clients are tightly coupled with the Proxy class and could become less reusable. Moreover, when for evolution purposes clients are allowed to directly access the RealSubject, they have to be modified to accommodate this change, i.e. each reference to a Proxy class should be changed to a RealSubject one[2]. Such changes have to be manually performed by the programmer each time client classes have to be adapted, for example also if the RealSubject has

[2]More details on such changes can be found in section 4.6.1.

to be shielded by a different Proxy class.

In such cases if the programmer uses one of the AODP versions no changes are needed in client classes. Clients are written to directly access the RealSubject class, instead of being tightly coupled with a specific Proxy shielding it. The Proxy behaviour is obtained by weaving the `ProxyPattern` aspect and annotating the RealSubject class. This makes client classes decoupled from a specific Proxy, thus they remain unchanged, e.g., if reused in a different application, or if the actual Proxy class changes or if a Proxy is added to (or removed from) a RealSubject. Even using a specialised aspect the same results can be obtained, as the generated aspect intercepts calls from the RealSubject and not from the Proxy.

As the previous discussion has shown, a programmer adopting the proposed AODPs can write better code with respect to the object-oriented alternative. Client classes can be designed to be independent of specific roles other classes play (such as the aforementioned Proxy example), e.g. such classes do not mix domain code with pattern-related code, thus producing simpler, more reusable classes.

Of the proposed AODPs, the AA and CA versions need some annotations to be added to involved classes for an aspect to be activated, either by means of a connector aspect or directly on the involved classes; such annotations are not needed for the specialised aspects. In both cases the benefits are very similar, as the previous discussion has shown. When designing a new application a programmer could adopt any AODP versions: the choice is essentially related to the different running times between the reflective and non-reflective alternatives (section 5.2).

Legacy object-oriented classes implementing a design pattern could also be used by performing some refactoring steps (section 4.6.1) prior to their integration in the system. After such classes are modified, both reflective (AA and CA) and non-reflective (SA) AODPs can be used. By using the reflective approach, classes playing specific roles for the design pattern have to properly be annotated, while in the SA approach this is not needed, as the generated aspect contains the specific pointcuts used to perform the expected pattern's behaviour.

The proposed AODPs have been designed to allow a better modularisation of the pattern code and its related classes. However, there are possible scenarios where they can't be adopted. For example the *Proxy* AODP can not be used as a *virtual proxy*, as the aspect's mechanisms can not avoid to create a RealSubject when a client instantiate it, thus the aspect can not defer its creation to the moment the *virtual* *proxy* would perform the instantiation. Another limitation is due to the generality of the used reflective technology, as in the AA and CA versions no type checks can be performed at compile time on the string parameters of the annotations, thus specific test suites may be adopted to ensure the final application not to throw any related runtime exception.

5.2 Performance assessment

The goal of the performance assessment study is to find out how much the proposed AODPs impacts the running times of an application, compared with the regular object-oriented alternative.

An application using the AODPs is expected to be slower than the object-oriented counterpart, with the execution overhead usually (and mainly) dependent on runtime checks and searches [FF05], and especially the AA version of a design pattern adds several reflective instructions to be executed in the normal program flow.

Another reason for the expected slower running times is the use of the aspect technology. The weaving of an advice into an application might impose some conditional instructions to be inserted and evaluated at runtime, as the pointcut can not evaluate them at compile time. For example the evaluation of `target` in a pointcut can not be determined at compile time as the target object exact type might only be determined at runtime, when the object is actually instantiated.

The approach for this assessment is to measure and compare just the design pattern's mechanisms running times between the implementations alternatives, not the whole running time of the application. To do so the *microbenchmarks* approach, and its related guidelines suggested in [FF05], has been used.

A microbenchmark measures the running time of a fragment of code, the measured fragments have been chosen to take into account the execution times of the instructions related to the pattern management.

The basic form of a microbenchmark is to put a timing instruction just before and after the fragment of code to be measured. As the running time of the fragment is expected to be small (with respect to the clock resolution of the machine running the tests), the fragment is repeated n times and the assumed running time is the average of the n repetitions.

Some of the suggestions followed from the guidelines are reported here and have been applied to all the presented measures. The measured code has been "warmed up" before the measure. I.e. the methods to be measured are executed before the measure itself so to allow all the classes to be loaded before the measure, so the class loading time is not counted in the running time. To avoid possible compiler optimisations to misrepresent the measures, the method body are as small as possibile, however not empty (to avoid the compiler to inline them), a public static variable is incremented, so, being accessible from outside, it can not be removed by the compiler. In the machine running the experiments[3] all the unessential operating system's services have been disabled to avoid interferences.

Each experiment was performed with n set to 1 million iterations and repeated 100 times, the resulting standard deviation of the measured running time for any experiment is under 5% of the average running time. The values shown in the following tables are all taken using these parameters.

The indexes (n, m) in the first column in table 5.1 tell the number of instances (n) and classes (m) playing the different roles for the related design pattern. In particular:

- *Flyweight*: n ConcreteFlyweights of m different classes. Measurement: a client requesting a (possibly new) ConcreteFlyweight.
- *Proxy*: n instances of Proxy, partitioned in m different classes (all implementing the same Subject interface). Measurement: invocation of a shielded method of a RealSubject.
- *Composite*: n Leafs of m different classes. Measurement: invocation of a method on a Composite object, iterated on all its Leafs.
- *Observer*: n ConcreteObservers of m different classes (all implementing the Observer interface). Measurement: invocation of an observed method which triggers the n ConcreteObservers to be updated.

The possible scenarios to measure for the *Singleton* design pattern are a subset of the *Flyweight* ones. This is due to the similarities of both pattern's behaviours and, thus, the proposed implementations. Both patterns have to return to the

[3]Apple MacBook, 2.26 GHz Intel Core 2 Duo with 4 GB RAM.

client the unique instance of an object of a given class, with the *Flyweight* one also recognising a limited number of different objects of the same class using a key. However, preliminary experiments have shown no significant overhead in running times of both patterns, hence just the *Flyweight* microbenchmarks are shown.

The microbenchmarks represent the complete execution time of the related method (for the object-oriented implementation) or the related method and the related advice (for the proposed aspect-oriented versions). The instruction used for the measurements is `System.nanoTime()` which, as the Java API [Sun07] states, "returns the current value of the most precise available system timer, in nanoseconds."

Table 5.1 reports all the microbenchmarks, expressed in μs, while table 5.2 reports the ratio between aspect-oriented and object-oriented versions. The SA version is presented for all the design patterns, just one of the reflective alternative is presented, as they just differ internally and have the same black-box behaviour.

Nearly every running time for any aspect-oriented version is longer than the object-oriented alternative, as expected. The AA version of *Flyweight* version is between 9 and 16 times slower than the object-oriented version, while the SA version provides better results being just between 4 and 5 times slower. The CA version of *Proxy* is between 77 and 766 times slower and generally decreases when m increases. However the SA version results bounded between just 5 and 27 times the object-oriented alternative. The *Observer* design pattern provides smaller running times increment in both the CA and SA versions, being respectively between 1 and 7, and 1 and 5 times slower[4]. The CA version of the *Composite* design pattern manages to be between 5 and 32 times slower than the object-oriented alternative, while the SA version is comparable with the object-oriented implementation.

Just by seeing these values, the overhead of the AODP approach might seem to hinder its applicability whatever modularity enhancement it might bring. However, the shown values can not directly tell how much an application using the AODPs will be slowed down. To understand the actual slowdown for the final application, it is possible to use a modified version of the Amdahl's law [HP06] used to compute the speedup of a CPU. It can be also be used to capture the slowdown of an application, such as the object-oriented and aspect-oriented ones, as described in [FF05] and briefly reported here for the sake of clarity.

The measured quantity for an aspect-oriented microbenchmark represents just

[4]Excluding the (1,1), (2,2) and (5,5) cases.

	Flyweight			Proxy				Observer				Composite			
n, m	OO	AA	SA	OO	AA	CA	SA	OO	AA	CA	SA	OO	AA	CA	SA
1,1	0.03	0.30	0.17	0.02	2.77	1.17	0.09	0.18	1.89	0.21	0.07	0.12	1.48	0.99	0.20
2,2	0.06	0.60	0.26	0.02	3.91	2.08	0.11	0.19	1.90	0.23	0.08	0.11	1.27	0.81	0.17
5,5	0.13	1.46	0.56	0.04	9.76	4.89	0.27	0.24	2.01	0.32	0.15	0.11	1.05	0.64	0.08
20,1	0.33	4.97	1.83	0.03	46.17	16.58	0.59	0.26	2.25	0.61	0.37	0.35	19.37	11.50	0.67
20,2	0.41	5.36	1.85	0.04	48.36	15.15	0.59	0.28	2.30	0.67	0.38	0.25	9.89	5.94	0.34
20,5	0.41	5.64	1.89	0.05	43.78	16.94	0.64	0.70	4.37	1.33	0.72	0.26	3.96	2.36	0.15
50,1	0.79	12.33	4.42	0.05	116.27	40.63	1.47	0.47	4.97	1.85	1.21	1.19	48.44	28.97	1.95
50,2	1.06	13.26	4.58	0.06	120.33	40.42	1.40	0.52	5.03	1.93	1.27	0.71	24.74	14.86	1.14
50,5	0.94	14.04	4.60	0.08	118.54	42.23	1.53	1.35	5.57	2.41	2.02	0.58	9.96	5.94	0.52
100,1	1.57	24.62	8.81	0.12	208.98	87.38	2.78	0.74	8.44	5.18	4.17	4.32	98.40	59.31	5.52
100,2	1.87	26.66	9.05	0.17	197.54	82.38	2.83	0.83	8.64	5.40	4.36	2.31	50.11	31.20	3.18
100,5	1.69	27.87	9.13	0.19	245.80	83.88	3.00	2.51	9.89	6.34	6.08	1.37	20.11	12.41	1.40

Table 5.1: *Microbenchmarks for the design patterns' versions (in µs)*

	Flyweight		Proxy		Observer		Composite	
n, m	AA/OO	SA/OO	CA/OO	SA/OO	CA/OO	SA/OO	CA/OO	SA/OO
1,1	9.09	5.06	77.8	5.87	1.15	0.39	8.21	1.63
2,2	9.6	4.26	104	5.7	1.19	0.42	7.38	1.56
5,5	11.28	4.33	135.75	7.61	1.32	0.6	5.72	0.68
20,1	15.28	5.62	592.04	21.18	2.33	1.41	32.59	1.91
20,2	13.07	4.5	420.89	16.44	2.37	1.33	23.39	1.35
20,5	13.94	4.66	319.68	12.02	1.92	1.04	9.08	0.59
50,1	15.63	5.6	766.66	27.79	3.91	2.56	24.44	1.65
50,2	12.54	4.34	696.97	24.19	3.73	2.45	21.08	1.61
50,5	14.94	4.9	563.11	20.35	1.78	1.49	10.19	0.88
100,1	15.68	5.61	753.23	23.98	7.04	5.67	13.73	1.28
100,2	14.26	4.84	473.42	16.25	6.52	5.26	13.52	1.38
100,5	16.54	5.42	446.16	15.96	2.53	2.42	9.04	1.02

Table 5.2: *Ratio of the microbenchmarks for the design patterns' versions*

the time it takes for the AODP to be activated and pass the control to the related method. E.g. in the *Observer* design pattern the time spent by the advice to cycle through all the attached ConcreteObservers and invoke the `update()` method on them, compared with the same operation performed by the object-oriented `notify()` method of the ConcreteSubject. The time spent inside the `update()` method is not counted in neither cases, just the activation mechanism is measured. Thus the complete application slowdown is proportional to the time spent by the application performing such pattern-related operations in the possible alternatives.

The slowdown is computed as follows

$$slowdown(x) = \frac{\frac{RTime}{NTime} + x}{1 + x}$$

where

- $NTime$ is the time spent for the execution of the regular, nonreflective, implementation (i.e. object-oriented version);
- $RTime$ is the time spent for the execution of the reflective alternative (i.e. aspect-oriented versions);

- x is the scaling factor representing how much time is spent in the application doing anything else; x is a multiple of $NTime$.

It is worth noting that the slowdown curve asymptotically approaches the $y = 1$ line.

Using the slowdown function it is possibile to understand the realistic impact on the running times of any application using AODPs compared with the same application using the object-oriented ones.

In the test machine on which the measures have been performed, a simple `println("Hello!  World")` instruction has been measured (also as a microbenchmark) and this takes 7.5 μs. Such information will be used as a reference to properly evaluate the slowdown caused by the AODPs.

With such a value, the actual slowdown for the application, shown in figures 5.1 and 5.2, is computed as a function of the scaling factor, i.e. the time spent by the application doing anything that is not pattern-management code. Such a scaling factor is represented on the x axis. On the y axis it is represented the value for the slowdown function. The parameters, $RTime$ and $NTime$, used to compute the curve are taken from table 5.2 for each scenario. Just a meaningful set of curves is shown.

Some of the values shown in table 5.2 are below 1, this happens for several SA versions. This means that such specialised aspects perform better than the object-oriented alternative in that scenario, as it can also be seen by comparing the values in table 5.1. The reason for such results is not completely unexpected, as the SA versions are, in practice, an object-oriented version rewritten using aspects, as no reflection is used in their code. Such values are excluded from both the following evaluations and the slowdown curves, as the focus of the following is just the slowdown.

The slowdown for the *Observer* in the CA version, with 20 instances of ConcreteObservers played by 5 different classes (i.e. row (20,5) in the table), the ratio $RTime/NTime$ amounts to 1.92. Thus for $x = 2$ the whole application is 30% slower than the object-oriented counterpart (as the slowdown amounts to 1.30), while with $x = 5$ it becomes just 15% slower (as the slowdown is 1.15). In practice, the x value can be compared with the time spent, for example, to perform the aforementioned print instruction on the same machine. A single `println` instruction is almost 11 times $NTime$ (7.5 μs/0.7 μs) for the CA *Observer* (20,5), thus for an application

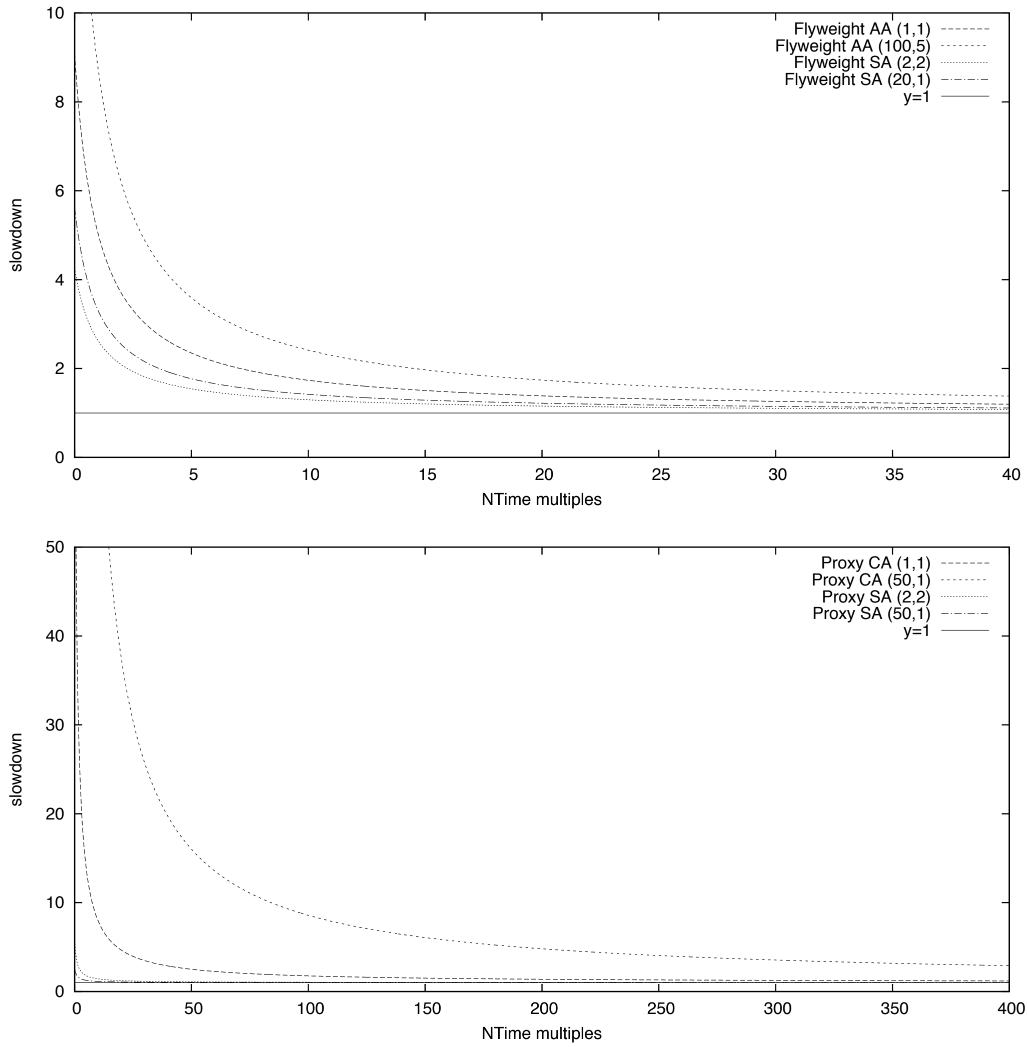

Figure 5.1: *Sample* slowdown *curves (*Flyweight *and* Proxy*)*

having a single execution of the said print instruction and the execution of the CA *Observer* (20,5), the application will have to bear an overall 7% slowdown with respect to the object-oriented alternative, i.e. $slowdown(11) = 1.07$. With ten print instructions the application is just under 1% slower than the object-oriented alternative. Similar evaluations can be formulated on all the slowdown curves shown.

The values just mentioned are different for any AODPs, as the different curves show. The ratio for the AA *Flyweight* design pattern ranges from 9.09 and 16.54, respectively for the (1,1) and (100,5) cases. In the former case a single print instruction brings the slowdown to just 3%, while in the latter it takes 100 print instructions to have the application be 15% slower. In the SA version the minimum ratio, 4.26,

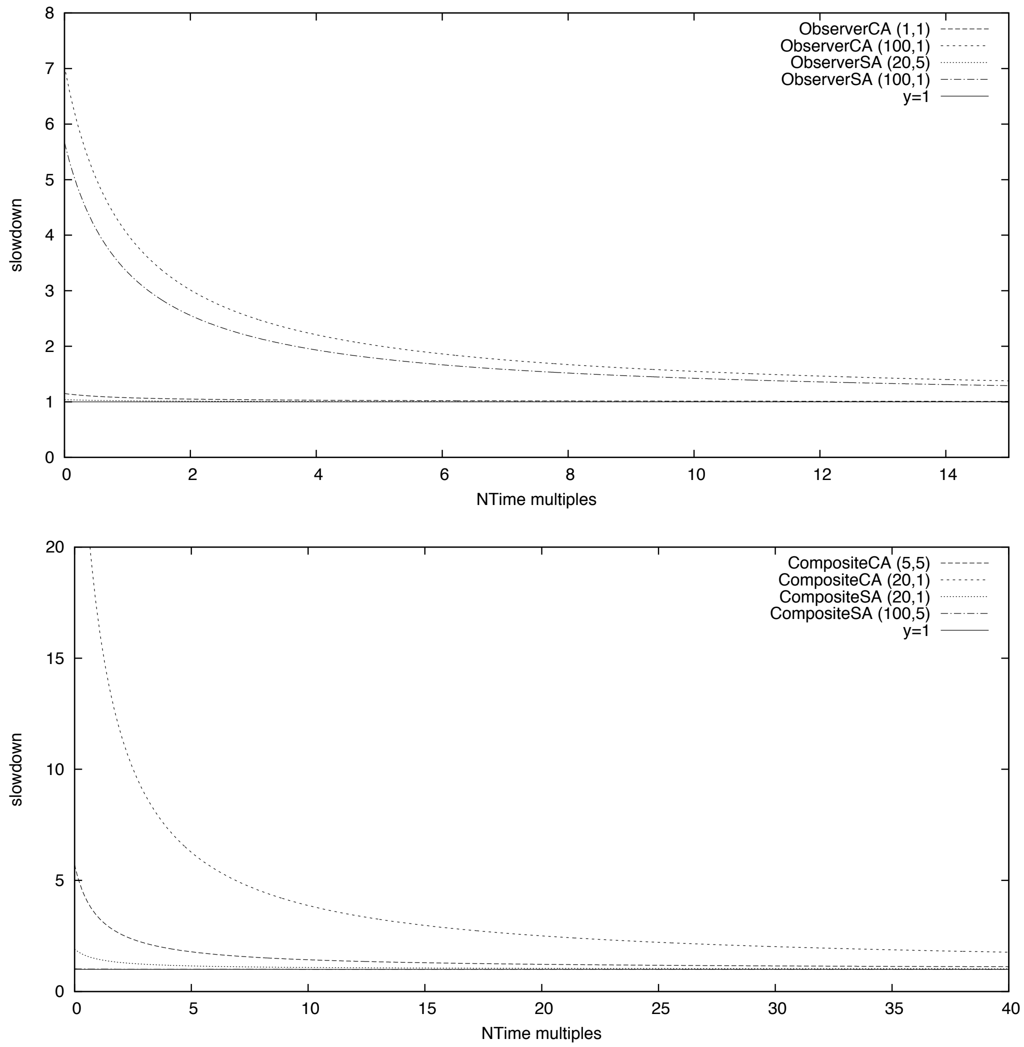

Figure 5.2: *Sample* slowdown *curves (*Observer *and* Composite*)*

brings a slowdown of just 2% with a single print instruction. With the maximum ratio, 5.62, just 50 prints lead the application to be just 9% slower.

The CA *Proxy* design pattern has a minimun ratio of 77.8 and a maximum of 766.66, respectively for the (1,1) and (50,1) cases. This pattern implementation is the slowest between all the proposed AODPs, however, 50 printing instructions are sufficient to have any application using it to be just 10% slower than the object-oriented alternative. In the SA case the ratio ranges between 5.7 and 27.79, in the worst case with just 2 `println` an application will be 8% slower, while in the best one just one print will make the application have a 1% slowdown.

The CA and SA *Observer*'s ratios are small enough to let just 10 `println` instructions to have an application be less than 5% slower in all the cases.

For the SA *Composite* just 2 `println` instructions make an application just 7% slower in the worst case; in the worst case for the CA scenario 20 `println` brings the same 7% slowdown.

Of course for a real application, and in general, the actual code outside design pattern's mechanisms will be much bigger than the few print instructions mentioned, hence the slowdown affecting the application is in practice much smaller.

The main result of these evaluations is that the actual impact on the running times of an application using the AODPs just looking at tables 5.1 and 5.2 might appear worse than it really is. The slowdown has to be evaluated as the time spent performing the pattern-related code with respect to the whole running time of the application. Just several simple printing instructions can practically make the proposed aspect-oriented versions run with the same running times as the object-oriented ones, with the added benefits of a better modularity. Moreover, improvements to mechanisms, e.g. handling dynamic invocations [Caz04], could be used in order to further reduce execution times.

Chapter 6

Related work

Design patterns have been widely studied and several proposals have been made to improve their modularity. This chapter reviews the literature on existing approaches and compares them with the aspect-oriented proposals of this dissertation, putting particular care in the analysis of the state of the art approach of Hannemann and Kiczales [HK02].

6.1 Hannemann and Kiczales' approach comparison

In the pioneering work from Hannemann and Kiczales [HK02] the 23 design patterns from [GHJV94] have been converted into an aspect-oriented version. Their approach proposes to separate the code implementing a design pattern into an abstract and possibly several concrete aspects. The abstract aspect of a design pattern usually defines utility interfaces and the basic behaviour of the pattern, e.g. for the *Observer* design pattern the iterative code for the `notify()` call on the observers. Such an abstract aspect is meant to be reusable, as its code does not mention any specific class. To be able to use the pattern it is however mandatory for the programmer to manually write at least a concrete aspect, which acts as a connector between the abstract aspect and the application. It maps the roles the actual classes play and usually adds additional code to concretise the abstract methods defined in the abstract aspect.

The first difference with the approach proposed in this dissertation is exactly about the reusability of patterns' implementation. Although the abstract aspect is absolutely reusable, no produced concrete aspect is. Any concrete aspect a programmer writes for a specific implementation can not be reused in other contexts as by its very nature it is tightly coupled with the specific application it has been written for. This causes a programmer to write very similar code[1] any time a design pattern has to be implemented in a different application, gaining just little benefits from the approach's usage. This is not the case in the proposal of this thesis, as the AA and CA versions of a design pattern are completely reusable as they need not additional gluing code to be attached to a particular application, thanks to the use of reflection. For the generated aspects of the SA approach, while they are not reusable, they can easily be automatically generated from a template.

It could be argued that this lack of generality in the aspects defined in [HK02] allows the authors to tackle all the regular design patterns, as the abstract aspect can simply define an abstract pointcut as a hook point with the application, delegating the responsibility of its actual definition to a concrete aspect (i.e., ultimately, to the programmer). However, for some design patterns, e.g. *Observer* and *Proxy*, most of work has to be implemented as a concrete aspect.

In the following, selected implementations from [HK02] are compared with the respective aspect-oriented versions of this dissertation.

Flyweight

The *Flyweight* version in [HK02] is organised in terms of abstract and concrete aspects, thus it has the already mentioned limitations about reusability and duplicated code. In this approach the FlyweightFactory role, instead of being implemented in a class, is implemented into a concrete aspect, thus all the clients need to explicitly call its methods to get a ConcreteFlyweight instance. This voids the *SPoC* property, as removing the ConcreteFlyweight role from a class implies to accordingly update all the clients' code accessing it. Moreover *REoR* is not satisfied, as the programmer could try to instantiate a ConcreteFlyweight directly (using `new`), bypassing the *Flyweight* instantiation logic and obtaining a new reference. This can not take place in the aspect-oriented versions proposed in this thesis, as the programmer would use the `new` constructor oblivious whether a class is playing the ConcreteFlyweight role.

[1] In several cases this is also duplicated code.

Proxy

A limitation of the *Proxy* implementation of [HK02] is that it does not discriminate between different RealSubjects, i.e. when two different RealSubjects of the same class are instantiated, the concrete aspect will use the same Proxy instance for both RealSubjects. The programmer is free to modify a concrete aspect to tell apart the different instances, however the aspect-oriented versions of this dissertation automatically performs the pairing with different Proxy objects. Thus [HK02] does not properly support the *REoR* property.

Another limitation is that it might not be so easy to instantiate a concrete aspect, as the programmer has to: (i) define the roles played by each class; (ii) define all the signatures of all the methods to be proxied; (iii) redefine some of the concrete aspect's methods. Thus concrete aspects might differ just in their mapping between classes and roles, i.e. the programmer has to replicate code for the concrete aspects (thus *ECoR* is not satisfied).

In the versions of the *Proxy* proposed in this thesis it is easier and less error-prone to define only the mapping with the roles by annotations, both directly or with a connector aspect (i.e. *SPoC* is not verified in [HK02]), while all the code for the pattern implementation is fully contained in the `ProxyPattern` aspect, instead of an abstract aspect and several concrete ones (i.e. *SoC* is not verified in [HK02]).

Observer

The abstract aspect proposed in [HK02] defines the Subject and Observer roles and contains the collection of ConcreteObserver instances, paired with the respective ConcreteSubjects.

To define a concrete aspect to be used in an actual application, the programmer has to: (i) define the (role, class) pairs; (ii) list all the observed methods' signatures which will trigger the updating logic; (iii) manually define the method to call by the updating logic. The concrete aspect is far from being reusable, especially compared with the proposed aspect-oriented versions of this dissertation which just use annotations and thus exempting the programmer from the aforementioned burdens.

Moreover, the *REoR* is not satisfied in [HK02], as the programmer could simply make a mistake and invoking the wrong class for a ConcreteObserver.

Composite

The abstract aspect containing the basic behaviour of the design pattern defines its roles as interfaces and provides several methods for managing the children's lists. It also generalises the operation to be performed using an implementation of the *Visitor* pattern, thus forcing the programmer to reason in terms of such pattern and use awkward code in the concrete aspect so to accommodate this design choice.

This imposition is not present in the aspect-oriented versions of this thesis, as the programmer can reason about the domain problem independently of another pattern, also the code of a role is thus contained in one place (the Composite class) instead of being separated in both the Composite class and the concrete aspect, i.e. the proposed versions yield a better *SoC* and satisfy *CDoR*.

6.2 Other approaches

Apart from the state of the art approach analysed in the previous section, the literature about design patterns presents many possible enhancements of the object-oriented paradigm that would also provide a better modularisation of design patterns, such as Composition Filters [AWB+94], the LayOM architecture [Bos99] and various software architectures [BMR+96, MWY91].

Examples of the categories of limitations of object orientation when dealing with design patterns' implementations are the pattern's traceability [Bos98, HB02], as the object-oriented host languages used for a design pattern implementation do not support them with a first class representation, and the reusability of the pattern's code. Arguably proposals such as Composition Filters and the LayOM architecture have been evolved in concepts also implemented with aspect orientation, and although studies such as [KAB07] reports on the difficulties of using aspect orientation in Feature Oriented Programming [Pre97], many authors put forward the use of aspect orientation in relation to design patterns implementations, such as [NK01, HK02, HB02, HLW03, BH07]. Selected studies are reviewed in the following.

In [NK01] the authors propose to use aspects to separate the code of design patterns from that of application classes. They put forward the use of aspect orientation as a way to improve the modularity of the resulting code. They propose to separate a design pattern's implementation into a reusable abstract aspect and one or more concrete aspect, the former to encapsulate the application-independent parts of the pattern, the latter to define the mapping with the actual application classes. Concrete aspects have to be manually coded, and as such they are not reusable and might need additional code to implement the desired pattern behaviour. Thus *SoC*, *SPoC* and *ECoR* are not verified.

In [HB02, HB03] the authors advocate the use of advanced separation of concerns techniques, especially aspect orientation, as a solution of what they consider the main cause for some problems arising in design patterns' implementations using object orientation, i.e. caused by code scattering and tangling. An important example of such problems is the traceability of an implementation, as the code of a design pattern results spread into different modules and thus hinders code readability and maintenance. Other problems they found are inheritance dependency and encapsulation breaching. The authors also stress the importance of an aspect-oriented description of design patterns in catalogues such as [GHJV94].

The authors put forward the use of a single aspect to implement a design pattern, so to enhance its traceability. Thus they provide the *Visitor* and *Strategy* implementation using a single aspect. However, such implementations, while improving the traceability of the design pattern's implementation, fail to be reusable as they are especially written for a specific application.

In [MF04], the authors analyse the approach in [HK02] finding some limitations affecting the proposed aspect-oriented version of design patterns. One of such limitations is the missing reusability of some of the produced aspects. A deeper experiment is performed in [MF08], where the authors start from an object-oriented implementation of the *Observer* design pattern and, using known refactoring steps, aim to derive the aspect-oriented version proposed in [HK02]. A result they find is that even in small applications the reuse of the abstract aspect of [HK02] is not trivial to exploit, as the authors state: "though the abstract aspect from [HK02] is potentially reusable, it had to undergo invasive changes in order to adapt it to the simple Java example [used]". However, the abstract aspect they derive from the refactoring steps is very similar to the original in [HK02], while the concrete aspect is tightly coupled with the application, thus retaining the issues already mentioned.

The author of [Can04] proposes the use of specific keywords as an extension to aspect-oriented languages so to allow a better language expressiveness when dealing with design patterns. Some of the proposed keywords are `roles`, `generic` and `class-set`. However a keyword like `multiple` seems useful just for a few design patterns. These can be used to define an abstract aspect as a base for the pattern implementation. Such an abstract aspect must be extended by a concrete one that defines the role–class mapping specific for the application. This reminds other approaches already analysed with the added limitation of forcing the programmer to deal with the specifically defined additional keywords. For this approach *SoC*, *ECoR* and *SPoC* are not satisfied, whereas *REoR* is difficult to evaluate, in that no implementation is provided of an environment supporting the proposed keywords. Moreover, the approach would be based on a non-standard weaver, making it less general.

The authors of [HU03] propose *parametric introductions* to allow the insertion of code fragments into classes, such additional code depends on parameters used in the weaving process. This approach is based, like the previous one, on specific extensions to be applied to the aspect-oriented language to be used, and thus providing the programmer with more powerful, non-standard, pointcuts. For example a `C` class that has to play the Singleton role forces to programmer to define a parametric aspect that inserts, via parametric introduction at weaving time, the `getInstance()` method into `C`. The classes affected by the introduction can be described in a concrete connector aspect thus making the abstract aspect reusable. The connector aspect might remind the aspect proposed in section 3.7, however the one they propose is based on a non-standard language and produces non reusable classes, while the AA and CA approaches use the connector aspect just to statically inject annotations into application classes, not to change their code, but to prepare them to (re)use the related aspects. *SoC* and *SPoC* are not satisfied, as the modified classes still mix their domain code with the design pattern one, also *ECoR* is not satisfied as the programmer could by mistake avoid using the `getInstance()` method introduced. Moreover all clients need to be updated when the `C` class should not play the Singleton role anymore.

In [KRH04] another extension of AspectJ is proposed. This one allows logic variables to be used to represent packages, types, fields and methods within "generic" aspects. The values for such logic variables are set by conditions' evaluation on join points, this is similar to the parameters in the parametric introductions approach already described, as the variable would ultimately be tied to actual application members. They implement the *Decorator* design pattern in a “generic” aspect which however still needs specific aspects to define roles played by application classes. *SoC*, *ECoR* and *SPoC* are not satisfied.

About the benefits of aspect orientation for design patterns implementation the authors of [GSF^{+}05] thoroughly studied the aspect-oriented implementations of [HK02] and obtained a quantitative assessment of the benefits of the aspect-oriented approach by comparing all the patterns of [GHJV94] in both object- and aspect-oriented fashion. The metrics used are extended versions of the classic object-oriented metrics [CK94] to be used for aspect-oriented implementations. Some aspect-oriented implementations resulted in more complex or coupled code than the object-oriented versions, however for a great variety of patterns, values for metrics such as coupling, cohesion and size are improved.

In [HMK05] an aspect-oriented refactoring approach for generic *crosscutting concerns* (cf. section 2.2) is put forward. The authors apply it to the special case of design patterns implemented as in [HK02]. The refactoring approach forces the developer to describe a refactoring (e.g. a design pattern occurrence in an object-oriented application to be changed into an aspect-oriented alternative) in terms of roles and relationships between roles. The description has to be defined by means of a non standard notation which uses keywords like `hasArgument` or `aggregates`. The authors test the approach to refactor an existing object-oriented application implementing several design patterns. Such patterns are converted in their equivalent aspect-oriented version as in [HK02], i.e. using an abstract aspect and possibly several concrete ones. Thus the implementation resulting from the said aspect-oriented refactoring approach would still have the same limitation of the implementations of [HK02].

How to adopt such an approach with a different modularisation of a design pattern, such as one of the versions presented in this dissertation, is not straightforward. E.g. whether their assumption of an abstract aspect representing the general pattern behaviour is mandatory or not. While some basic refactoring steps they proposed could remain unchanged, others might be changed to accommodate different design pattern structures. E.g. the *Observer* of [HK02] needs to define a Subject interface which is not needed in the AODP version presented in this thesis, thus would lead to a different formulation for the refactoring steps of the same pattern (implemented in a different fashion). Further studies might lead to an extension of their approach to also encompass AODPs. Such an extension, however, would just be limited to a different way of refactoring for the SA approach.

The authors of [FF05] put forward a generative approach for design patterns, however just by using object orientation and computational reflection. Their approach extends a class, say `C`, on which a design pattern's role has to be imposed, by subclassing `C`. The automatic generation of a subclass of `C` is performed by introspecting the original class. For example in the *Singleton* design pattern the generated subclass would have a generated private constructor which the clients is expected to use. Such generation would however produce object-oriented implementations, so it would not appropriately satisfy *SoC* as the final class would mix both domain and pattern related code. The *REoR* property is not satisfied as a programmer could also explicitly invoke a `new` on the original class. Moreover, the *SPoC* property would not be satisfied as all the clients are tightly coupled with the generated subclass. E.g. clients are expected to invoke the static method on the subclassed Singleton class and have to be accordingly changed when such class does not play the role anymore.

Other approaches, loosely related to aspect orientation, have been proposed and could be used for design patterns' implementations, with Object Teams [Her03] and CaesarJ [AGMO06] being two notable examples.

In Object Teams an *object team* defines a set of collaborating classes (called *roles*), variables and methods in a single module. With the *callout* mechanism, a team can superimpose a *role* to a base class, i.e. mapping a method from a *role* class to the base class. This allows the invocations of the method of a role class to be redirected to the correspondent base class' method.

The *callin* mechanism is similar to a pointcut in aspect-oriented languages, as it allows a method of a role class to be called before, after or instead of the correspondent method on the base class. This is defined by mapping a method on the role class to a method on a base class. Even if similar to some AspectJ constructs, the mapping is defined according to the method's name and thus can not be reused.

A *Proxy* implementation using this language would use the *callin* to redirect the calls for each method of a RealSubject class to the related Proxy one, however it would require this method mapping to be manually expressed for each pair of methods, yielding non-reusable code coupled with such specific classes.

The CaesarJ language [AGMO06] uses the idea of *family class* (or *cclass*) that groups a set of collaborating classes, it also provides a join point language with the syntax in common with AspectJ.

A design pattern can be implemented writing an abstract *cclass* defining the roles for the pattern and additional managing code, then a concrete *cclass* can be defined to connect it to actual classes on which the roles have to be superimposed. The authors of [SM08] compared several implementations of design patterns in AspectJ and CaesarJ. The results are not conclusive enough but slightly leans towards the use of CaesarJ, however the implementations suffers from the same limitations noted for the [HK02] approach, even using the specific mechanisms offered by CaesarJ, as the reusable components have to be connected to the actual application classes thus producing non reusable code as in [HK02].

A declarative metaprogramming approach is presented in [MT01] and used to generate specific design pattern code for an application and to manage its evolution and refactoring also by generating code. The proposed framework is based on a variant of Prolog which uses predicates to represent object-oriented constructs. The target language is Smalltalk. E.g. `class(?C)` is used to state that `C` represents a class, and `abstractMethod(?C, ?M)` to state that method `M` of the `C` class must be abstract.

Such predicates are used by the programmer to generate the code for a design pattern, check for constraints to be verified by the pattern (such as inheritance relationships) and to perform refactoring transformations. To perform such tasks the programmer has to write several lines of Prolog code. In the case of generated classes for a design pattern the programmer has to include additional fragments of code to complete the pattern implementation. This is different from the SA aspects proposed in this dissertation, as, once generated, they do not need any further adjustments.

Moreover, the generated Smalltalk code does not verify properties such as *SPoC*, as when a design pattern needs to be removed all its client classes have to be updated, and *REoR*, as, e.g. for the *Singleton* design pattern, a programmer could also create a new object instead of invoking the `getInstance()` method.

Chapter 7

Conclusions

This dissertation presented a novel implementation of some design patterns, which allows to avoid some common problems that arise using object orientation.

Such implementations allow a design pattern to be compactly defined in a single, completely reusable aspect, without any further specialisation needed, thanks to the use of computational reflection. This is a step forward from the state of the art approaches. Indeed these force the programmer to separate a design pattern into an abstract and, possibly more than one, concrete aspect, of which only the abstract one is reusable.

An application employing the proposed versions can use the aspect implementing a design pattern as a module inserted into (or removed from) the application by simply adding (or removing) the aspect. The behaviour of a role played by a class in a design pattern is provided by the aspect and activated by annotations, which can directly mark involved classes or can be collected in a connector aspect. In standard object orientation practice, when a class is evolved to play a role for a design pattern, usually other application classes need to be updated to make use of such change. Instead, using the proposed approach, no other application classes need to be updated, as there is no coupling between them and the role-implementing classes.

The provided implementations also offer a better separation of concerns. A class just contains its domain code, instead of mixing it with the code implementing some design pattern behaviour, which is fully included in the aspect. Such classes are reusable and easier to evolve, as they are not concerned with additional code unrelated to their main responsibility.

The code implementing a desing pattern is not dispersed in different classes, but it is instead accessible from a single module. This makes it easy for the programmer to understand the presence of the pattern in the application and to easily discover involved classes by simply reading the connector aspect.

Another important property of the proposed implementations is the enforcing of a role behaviour for a design pattern. By means of aspect orientation the additional behaviour of a class for a given role is automatically enforced by activation of the advices, preventing errors from the programmer such as forgetting to update observers about a state change in an observed class.

Up to two different variants have been provided for each design pattern: the cached version, which avoids as much as possible repeating the execution of computational reflection methods, and the specialised, generated, version which does not use reflection at all. Moreover, specialised aspects do not make use of annotations as they can be generated for given classes.

All versions provide the same behaviour, each version yielding different running times for an application using it. Such running times have been extensively compared with the corresponding times obtained using standard object-oriented implementations. In many cases the aspect-oriented design patterns have been found comparable, and thus convenient to use, both for the modularity they bring and their performance.

The presented aspects can be used since the beginning of the design phase of an application, however a refactoring approach has also been put forward to make them applicable in legacy object-oriented applications. Such legacy applications, once refactored, can use any version of the proposed implementations.

A drawback of the proposed approach is its non-trivial extension to other design patterns. Indeed, it has been proposed just for a subset of the most common design patterns, but as a future line of research it would be interesting to further the study of such implementations for other design patterns. It would also be interesting to investigate how to find a general method to convert any object-oriented design patterns' implementation into its aspect-oriented version.

Bibliography

[AG96] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. *Transactions on Aspect-Oriented Software Development*, 3880, 2006.

[AWB+94] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting object interactions using composition filters. In *Proceedings of the Workshop on Object-based Distributed Programming of ECOOP'93.* Springer, 1994.

[BH07] Marc Bartsch and Rachel Harrison. Design patterns with aspects: A case study. In *Proceedings of the Workshops of 12th European Conference on Pattern Languages of Programs (EuroPLoP)*, 2007.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – A System of Patterns.* Wiley, 1996.

[Bos98] Jan Bosch. Design patterns & frameworks: On the issue of language support. In *Proceedings of the Workshops on Object-Oriented Technology of ECOOP'97.* Springer, 1998.

[Bos99] Jan Bosch. Superimposition: A Component Adaptation Technique. *Information & Software Technology*, 41(5), 1999.

[Can04] Jordi Alvarez Canal. Parametric aspects: A proposal. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolu-*

tion (RAM-SE) of ECOOP'2004. Fakultät für Informatik, Universität Magdeburg, 2004.

[Caz04] Walter Cazzola. Smartmethod: an efficient replacement for method. In *Proceedings of the 2004 ACM symposium on Applied computing (SAC)*. ACM, 2004.

[Chi00] Shigeru Chiba. Load-time Structural Reflection in Java. In *Proceedings of ECOOP*. Springer, 2000.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6), 1994.

[DZS09] Jing Dong, Yajing Zhao, and Yongtao Sun. A matrix-based approach to recovering design patterns. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 39(6), 2009.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Fer89] Jacques Ferber. Computational reflection in class based object oriented languages. In *Proceedings of Object-oriented programming systems, languages and applications (OOPSLA)*. ACM, 1989.

[FF05] Ira Forman and Nate Forman. *Java Reflection in Action*. Manning Publications, 2005.

[GHJV94] Eric Gamma, Richard Helm, Ralph Johnson, and Richard Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GMPT09a] Rosario Giunta, Fabrizio Messina, Giuseppe Pappalardo, and Emiliano Tramontana. Analysing the performances of grid services handling job submission. In *Proceedings of the 18th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE)*. IEEE, 2009.

[GMPT09b] Rosario Giunta, Fabrizio Messina, Giuseppe Pappalardo, and Emiliano Tramontana. Improving the performances of a grid infrastructure by means of replica selection policies. In *Proceedings of Final Workshop of GRID projects, "PON Ricerca 2000-2006, Avviso 1575"*. Consorzio COMETA, 2009.

[GMPT09c] Rosario Giunta, Fabrizio Messina, Giuseppe Pappalardo, and Emiliano Tramontana. Measuring performances of globus toolkit middleware services. In *Proceedings of Final Workshop of GRID projects, "PON Ricerca 2000-2006, Avviso 1575"*. Consorzio COMETA, 2009.

[GPT10] Rosario Giunta, Giuseppe Pappalardo, and Emiliano Tramontana. Using aspects and annotations to separate application code from design patterns. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC)*. ACM, 2010.

[GPT11] Rosario Giunta, Giuseppe Pappalardo, and Emiliano Tramontana. Aspects and annotations for controlling the roles application classes play for design patterns. In *Proceedings of the 18th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2011. *to appear*.

[GPT12a] Rosario Giunta, Giuseppe Pappalardo, and Emiliano Tramontana. AODP: Refactoring code to provide advanced aspect-oriented modularization of design patterns. In *Proceedings of the 27th ACM Symposium on Applied Computing (SAC)*. ACM, 2012. *to appear*.

[GPT12b] Rosario Giunta, Giuseppe Pappalardo, and Emiliano Tramontana. Superimposing roles for design patterns into application classes by means of aspects. In *Proceedings of the 27th ACM Symposium on Applied Computing (SAC)*. ACM, 2012. *to appear*.

[GSF$^+$05] Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing design patterns with aspects: a quantitative study. In *Proceedings of the 4th international conference on Aspect-Oriented Software Development (AOSD)*. ACM, 2005.

[HB02] Ouafa Hachani and Daniel Bardou. Using aspect-oriented programming for design patterns implementation. In *Proceedings of the Workshop on Reuse in Object-Oriented Information Systems Design of Object-Oriented Information Systems (OOIS)*, 2002.

[HB03] Ouafa Hachani and Daniel Bardou. On aspect-oriented technology and object-oriented design patterns. In *Proceedings of the Workshop on Analysis of Aspect-Oriented Software of ECOOP*, 2003.

[Her03] Stephan Herrmann. Object teams: improving modularity for crosscutting collaborations. In *Objects, Components, Architectures, Services, and Applications for a Networked World – Proceedings of the International Conference NetObjectDays (NODe 2002)*. Springer, 2003.

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 2002.

[HL95] Walter L. Hürsch and Cristina Videira Lopes. Separation of concerns. Technical Report NU-CCS-95-03, Northeastern University, Boston, 1995.

[HLW03] Robert Hirschfeld, Ralf Lämmeland, and Matthias Wagner. Design patterns and aspects – modular designs with seamless run-time integration. In *Proceedings of the 3rd Workshop on Aspect-Oriented Software Development (AOSD-GI) of the SIG Object-Oriented Software Development*. German Informatics Society, 2003.

[HMK05] Jan Hannemann, Gail C. Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In *Proceedings of the 4th international conference on Aspect-Oriented Software Development (AOSD)*. ACM, 2005.

[HOU03] Stefan Hanenberg, Christian Oberschulte, and Rainer Unland. Refactoring of aspect-oriented software. In *Objects, Components, Architectures, Services, and Applications for a Networked World – Pro-*

ceedings of the International Conference NetObjectDays (NODe 2002). Springer, 2003.

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach (4th ed.)*. Morgan Kaufmann, 2006.

[hp11] AspectJ home page. AspectJ, 2011. http://www.eclipse.org/aspectj.

[HU03] Stefan Hanenberg and Rainer Unland. Parametric introductions. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*. ACM, 2003.

[IZ03] Masanori Iwamoto and Jianjun Zhao. Refactoring aspect-oriented programs. In *The 4th AOSD Modeling with UML Workshop*, 2003.

[KAB07] Christian Kästner, Sven Apel, and Don Batory. A case study implementing features using AspectJ. In *Proceedings of the 11th Software Product Line Conference (SPLC)*. IEEE, 2007.

[Ker04] Joshua Kerievsky. *Refactoring to patterns*. Addison-Wesley, 2004.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Object-Oriented Programming – Proceedings of ECOOP'97*. Springer, 1997.

[KRH04] Günter Kniesel, Tobias Rho, and Stefan Hanenberg. Evolvable pattern implementations need generic aspects. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE) of ECOOP'2004*. Fakultät für Informatik, Universität Magdeburg, 2004.

[Lad09] Ramnivas Laddad. *AspectJ in Action: Enterprise AOP with Spring Applications*. Manning, 2nd ed., 2009.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA)*. ACM, 1987.

[MF04] Miguel P. Monteiro and João M. Fernandes. Pitfalls of AspectJ implementations of some of the gang-of-four design patterns. In *Proceedings of Iberian workshop on Aspect Oriented Software Development (DSOA04)*, 2004.

[MF08] Miguel P. Monteiro and João M. Fernandes. An illustrative example of refactoring object-oriented source code with aspect-oriented mechanisms. *Software: Practice and Experience*, 38, 2008.

[MT01] Tom Mens and Tom Tourwé. A declarative evolution framework for object-oriented design patterns. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2001.

[MWY91] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'91)*. Springer, 1991.

[NK01] Natsuko Noda and Tomoji Kishi. Implementing design patterns using advanced separation of concerns. In *Proceeding of the Workshop on Advanced Separation of Concerns in Object-Oriented Systems of OOPSLA*, 2001.

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *Object-Oriented Programming – Proceedings of ECOOP'97*. Springer, 1997.

[Pre05] Roger S. Pressman. *Software Engineering: A Practitioner's Approach, 6th ed.* McGraw-Hill, 2005.

[Pro11] Eclipse Project. Eclipse IDE, 2011. http://www.eclipse.org/eclipse.

[PT06] Giuseppe Pappalardo and Emiliano Tramontana. Automatically discovering design patterns and assessing concern separations for applications. In *Proceedings of the 2006 ACM Symposium on Applied computing (SAC)*. ACM, 2006.

[SM08] Edgar Sousa and Miguel P. Monteiro. Implementing design patterns in caesarj: an exploratory study. In *Proceedings of the 2008 AOSD* *workshop on Software engineering properties of languages and aspect technologies*. ACM, 2008.

[Som01] Ian Sommerville. *Software Engineering*. Addison-Wesley, 2001.

[SSRB00] Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[Sun07] Sun Microsystems. Java 1.5 documentation, 2007. http://java.sun.com/j2se/1.5.0.

[SW08] Krysztof Stencel and Patrycja Węgrzynowicz. Implementation variants of the singleton design pattern. In *Proceedings of On the Move to Meaningful Internet Systems (OTM)*. Springer, 2008.

[TCSH06] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11), 2006.

[WS00] Ian Welch and Robert J. Stroud. Kava – A Reflective Java Based on Bytecode Rewriting. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering: Reflection and Software Engineering, Papers from OORaSE 1999*. Springer, 2000.