Aspect Oriented Implementation of Design Patterns using Metadata

Andrei Oprisan

May 11th 2008
Master's Thesis
University of Joensuu
Department of Computer Science and Statistics

Abstract

Computer programming paradigms aim to provide better separation of concerns. Aspect oriented programming extends object oriented programming by managing crosscutting concerns using aspects. AspectJ is the most successful aspect oriented implementation. It extends the Java programming language with constructs specific to aspect oriented programming. Two of the most important critics of aspect oriented programming and AspectJ are the “tyranny of the dominant signature” and lack of visibility of program's flow. Metadata, in form of Java annotations, is a solution to both problems. Design patterns are the embodiments of best practices in object oriented design. Aspect oriented programming can be used to implement the most known patterns, the “Gof” patterns, in order to analyze the benefits. This thesis presents the results of using aspect oriented programming and metadata to implement the “Gof” patterns. The most successful implementations are the ones in which the pattern-related code crosscuts across the concerns encapsulated in the participants in the pattern. Successful implementations share a generic solution: the usage of annotation to configure and mark the participants, while the pattern's code is encapsulated in aspects. This looses the coupling between aspects and type signatures and between the code base and a specific AOP framework. Also, it increases the developer's awareness of the program's flow. The patterns are plugged/unplugged based on the presence/absence of annotations.

Keywords: Aspect Oriented Programming, Design Patterns, Metadata, AspectJ, Object Oriented Programming

Acknowledgments

I wish to thank my supervisor Dr. Simo Juvaste for his help, guidance and support in the making of this thesis.

I would also want to thank all the people previously and currently involved in the IMPIT program for giving me the opportunity to live such an unforgettable experience.

I am grateful to all my colleagues at Blancco Oy for showing me how it is to work in a professional environment, for their help and advices and for offering me the opportunities to evolve as a software developer.

I would like to thank all my colleagues and friends from Joensuu and home for their support, help and the good moments we had together.

I wish to thank my family for their understanding, help and support provided during my studies.

A special thanks goes to my dear wife Lili, for her endless patience, support, encouragement and the wonderful trips she organized for us in the moments when I took a break from writing this thesis.

Table of Contents

List of Figures

1. Introduction

As pointed out by Elrad et al. in [Elrad01], there is an evolution of programming languages, from assembly languages, through procedural programming, functional programming to programming with abstract data types. All of these concepts aim at providing better separation of concerns at source code level. The term separation of concerns was coined by Dijkstra in [Dijkstra82] and it is a design principle which promotes the parting a computer program intro distinct entities or features, overlapping as little as possible in functionality. A concern is a feature or behavior of the program. Separation of concerns can be achieved in different ways, one being using language constructs. As an example, Object Oriented Programming separates concerns intro classes and objects. It handles well the separation of the applications' logic concerns, but does little to accommodate the separation of crosscutting concerns. Crosscutting concerns are, as the name expresses, behavior that cuts across other concerns, usually not being a part of the application's logic. Aspect Oriented Programming [Kiczales97] (AOP) appeared as a response to the need of encapsulating crosscutting concerns. It is not meant to be a replacement for object oriented programming but rather an extension to it (though there are Aspect Oriented Programming implementations for procedural languages like C [AspectC]). AOP comes in different flavors and shapes for a large number of programming languages. A detailed description of AOP and AOP implementations will be presented in Chapter 2 of the thesis.

A pattern is a reusable solution to a problem that appears often in the domain of software design. They are to found for the first time as an architectural concept in the work of C. Alexander [Alexander77]. After ten years since the first edition of [Alexander77], the first results of experimenting the application of patterns to programming were published in [Beck87]. But not only after [Gamma95] was published that design patterns gained popularity among programmers. [Gamma95] is considered to be one of the essential books on software engineering and the reference book for design patterns, being mostly known as the "Gang of Four" book or, shorter, and “Gof" book. One of the greatest achievements of [Gamma95] is the creation of a common vocabulary, facilitating the communication between software engineers. By having their roots in architecture, one can say that design patterns helped software engineering to make a further step in becoming recognized as a true engineering discipline.

The software industry is in a continue search for new solutions. Every promising newcomer generates more or less hype around it. Gartner Inc. presented a trend in hype, so called Hype

Cycle [Gartner95], graphically represented in Figure 1. This model is a good example of adoption issues.

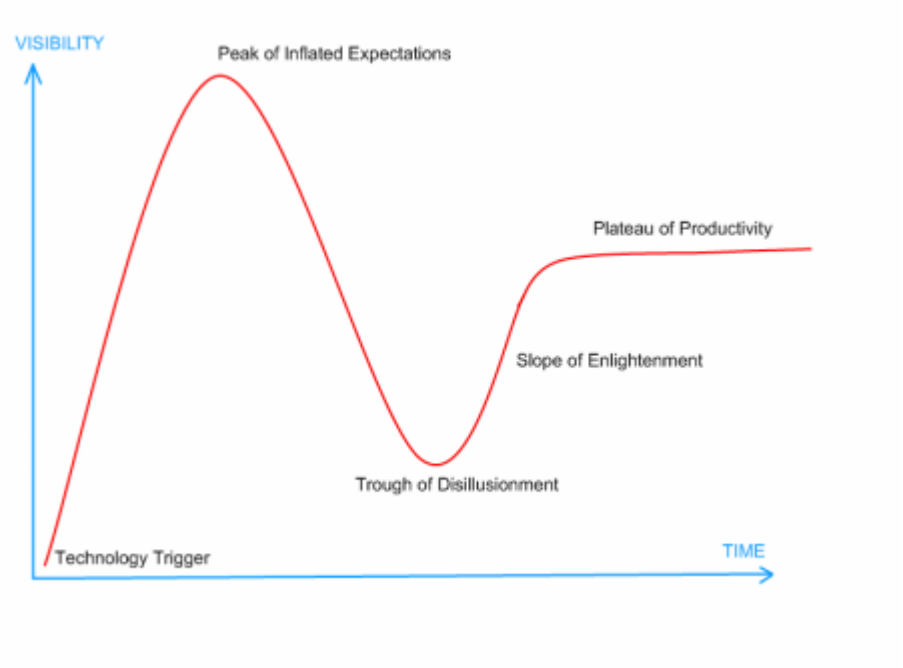

Figure 1: Hype Cycle

As it can be noticed in the figure, the moment when a technology can be dismissed is when the "Trough of Disillusionment" is reached. Aspect oriented programming was in the "Through of Disillusionment" and now is slowly moving up on the "Slope of Enlightenment". The biggest issues of AOP, and of AOP implementations, are: it is difficult to grasp by average a programmer; it influences the program's semantics and flow without the developer's knowledge, creating hidden bugs; high dependency on the names of program's artifacts.

The goal of the thesis it to make a contribution in this area by providing solutions for solving two critics of AOP, namely dependency on the names of classes, methods or fields; and lack of flow visibility. The solutions are built using an approach combining metadata and AOP. The structure of the thesis is presented in the following paragraph. Chapter 2 provides an introduction to AOP, presenting its concepts: joinpoint, pointcut, advice, and aspect. It also includes a brief presentation of the current AOP implementations, how they are classified and how they achieve their purpose. Chapter 3 contains a presentation of the principles behind the "Gof" patterns, an enumeration of the patterns and short description for each of them. The chapter also includes a discussion about the AOP implementations of the patterns that show limited or no benefit from using aspect orientation. Chapter 4 gives a detailed description of the patterns chosen as gaining

most from AOP and metadata: Singleton, Observer, State, and Proxy. Conclusions and further research directions are drawn in Chapter 5.

2. Aspect Oriented Programming

The emergence of the Aspect Oriented Programming (AOP) paradigm is driven by the need for better ways of describing and encapsulating concerns in a software application. Object Oriented Programming (OOP) provides a good way for this by using objects that encapsulate state and actions; however this is limited to the problem domain of an application. The so-called crosscutting concerns could not be fitted. Among the usual crosscutting concerns are logging, authentication and transaction management. These aspects are not related with the problem domain of the application but rather they "cut through" it. The current crosscutting concerns management is to interleave them with the core logic code. Unfortunately this breaks the modularization of the system. To solve this situation, research explored how crosscutting concerns can be isolated from the business logic and be applied in a non-intrusive manner. AOP was coined by G. Kiczales and his team at Xerox PARC in the early 1990's. Also, they developed one of the first and most popular AOP languages, AspectJ [AspectJ], as an extension to Java. AOP gained notoriety among software developers and architects, as systems have become more complex and old paradigms have been unable to keep pace. AOP does not replace OOP but extends it by providing further separation of concerns.

This chapter consists of two sections. The first one presents the generic concepts specific to aspect oriented programming. The second section contains a discussion about the classification criteria applied to aspect oriented programming implementations together with the brief presentations of several AOP frameworks: AspectJ, JBoss AOP and Spring AOP. This second section is by no means an exhaustive presentation of AOP frameworks, but an example of how AOP concepts are implemented in different approaches.

2.1 Aspect Oriented Programming Concepts

AOP achieves separation of concerns by providing a new unit of modularization, namely an *aspect* that crosscuts other modules [Laddad03]. Aspects have to be composed, a *weaving* process, with other modules in the system. This is achieved using a compiler like entity named an *aspect weaver* [Laddad03]. An aspect weaver can accomplish two types of crosscutting: dynamic crosscutting and static crosscutting. Dynamic crosscutting represents the injection of code (behavior) at certain points in the execution of the program, altering the dynamic part of the program, namely its execution. Static crosscutting is the modification of the static part of the

system (e.g. classes, interfaces). It has to be mentioned that static crosscutting is seldom supported by AOP implementations. An aspect encapsulates both dynamic and static crosscutting constructs. They are presented as it follows:

a) Joinpoint

A *joinpoint* is a conceptual entity defining the points in the execution of the software where crosscutting actions can be woven in. A joinpoint can be the assignment of a value to a variable, a method call or a constructor call. The multitude of joinpoints that can be captured is specific to each and every AOP implementation.

b) Pointcut

A *pointcut* is a construct that allows the specification of several joinpoints. It may also offer the possibility to collect the context for the joinpoints. How pointcuts are implemented and how much context they can collect, if any, is also specific to each and every AOP implementation. One can think of pointcuts as *weaving rules* and of joinpoints as places in the program flow where the rules are satisfied [Laddad03]. It is a dynamic crosscutting construct.

c) Advice

An *advice* is a construct which consists of two entities: a crosscutting action and the pointcut where the action should be applied (woven in). If the pointcut captures context, it has to be made available to the crosscutting action. It is a dynamic crosscutting construct.

d) Introduction

Introduction is a static crosscutting construct that performs static changes (modifying the inheritance tree, adding methods and members to classes) to the structure of other modules in the system.

e) Aspect

An *aspect* is the building block of AOP as the class is the building block of OOP. It encapsulates pointcuts, advices and introductions in one unit.

2.2 Aspect Oriented Programming Implementations

There are several AOP implementations available being the result of different research directions. Most of them are academically developed, while some are the result of industry involvement. The

Aspect Oriented Software Development official website [AOSD] provides an extensive list of implementations; though a significant part of them have stopped being developed (Aspect# [Aspect#]). The most widely used AOP implementation is Spring AOP, a part of the Spring Portofolio [Spring]. The Spring framework [Spring] evolved from being a lightweight dependency injection [Fowler04] framework to offering a complete suite of services for the development of java enterprise applications. Rod Johnson and Juergen Hoeller, the lead developers of Spring, introduced it as a solution to the verbosity of J2EE [JEE] in [Johnson04]. They illustrate the principles that allowed designing and building a lightweight approach to the development of enterprise applications. The most important techniques presented are dependency injection and AOP. Spring AOP [SpringAOP] is the AOP implementation included in Spring Portofolio. Its API was replaced in post 2.x Spring versions with the richer one offered by AspectJ, thus bringing AspectJ into the enterprise. The AOP features of Spring can be used without AspectJ, though some capabilities will be restricted.

When an AOP implementation is evaluated, two key features have to be observed: the moment when weaving occurs and how AOP constructs are expressed. There are three moments when the weaving can happen: compile time, load time and run time. They will be described as it follows.
Compile time weaving requires the aspect weaver to behave as a compiler. AspectJ uses compile time weaving, a detailed description of the process being presented in [Laddad03]. A simple description of the process is as the following: the aspect weaver reads the declaration of aspects, transforms the source code accordingly and the code is compiled using a standard compiler. After the transformation there is nothing left but ordinary code. Due to full access to the source code, usually the pointcut language is very rich. An important advantage to other types of weaving is speed; the woven application having no run time performance penalty because the compiled code is normal code.

Load time weaving, implemented as from AspectJ5 allows weaving aspects when a class is loaded in the virtual machine. It implies a performance penalty due to the byte code generation that takes place when a class is loaded, while simultaneously offering the full power of AspectJ's rich pointcut language.

Run time weaving is based on proxies. Classes to be touched by the aspects are hidden behind proxies containing the advice code. It implies a performance penalty as instances are not accessed directly but rather through proxies. Also, the pointcut language is not rich as in the case of compile time weaving implementations.

An AOP implementation can be developed as an extension for a language or as a framework. AspectJ will be presented as an example for the first approach while JBoss [JBossAOP] and Spring AOP (pre 2.x) as an example of the latter.

AspectJ, for expressing the concepts of AOP extends the Java language with keywords such as "aspect," "pointcut," "advice" etc. An aspect is the equivalent of a class in Java. It encapsulates pointcuts and advices. A pointcut is a sort of Regular Expression that matches one or more join points (E.g. **execution**(**void** Account.credit(**float**))). This pointcut matches the executions of the credit method of class Account that has a float parameter and a void return type. An advice is the equivalent of a method for the aspect. It needs a pointcut to be specified and also where should be applied: before, after, or before and after the methods captured by the pointcut. The following is a case example:

```
before() : execution(void Account.credit(float)) {
    System.out.println("before performing credit operation");
}

Object around() : execution(void Account.credit(float)) {
    Object result = null;
    System.out.println("about performing credit operation");
    result = proceed();
    System.out.println("after performing credit operation");
    return result;
}


after() : execution(void Account.credit(float)) {
    System.out.println("after performing credit operation");
}
```

The syntax makes it easier for a Java programmer to employ it, rather than learning how to write XML documents. Since AspectJ5, annotations where introduced so that aspects can now be declared as annotated java classes, making AspectJ seamlessly integrate with the Java language.

JBoss AOP and Spring AOP

Due to similarities in the approach used by both frameworks they are presented together. JBoss is considered to be the most popular open source J2EE Application Server. Both frameworks use XML for configuration, making the choice of XML for pointcut definition a natural one. Below is an example of a pointcut definition for JBoss:

```
<?xml version="1.0" encoding="UTF-8"?>
  <aop>
    <bind pointcut="execution(public void
                      aop.jboss.Order->addItem(java.lang.String,int))" >
      <interceptor class="aop.jboss.TraceInterceptor" />
    </bind>
```

```
</aop>
```

For writing the advices, just a normal Java class is needed that implements the “Interceptor” interface.

For a pointcut definition for the Spring framework, the following is a good example:

```
<bean id="tracePointcut"
            class="org.springframework.aop.support.Perl5RegexpMethodPointcut">
  <property name="pattern">
    <value>aop.spring.Order.addItem</value>
  </property>
</bean>
```

The bean identified by “tracePoincut” is the pointcut. The pattern is the expression of the pointcut. For writing interceptors, a normal Java class that implements TraceInterceptor is needed.

2.3 Conclusions

AOP extends OOP in order to provide mean for encapsulating crosscutting concerns. In order to do so, it adds its own set of concepts: joinpoint, pointcut, advice, introduction, aspect. Applying aspects to a codebase bears the name "weaving". The entity in charge of this process is called "aspect weaver". There is no aspect oriented programming language, AOP being present in the software engineering world in the form of frameworks. An AOP framework has two important components: a specific language to express AOP concepts and an aspect weaver. Hence, AOP frameworks can be classified according to these two components. The most important criterion is when weaving occurs: compile time, load time or runtime. This also has an impact on the set of AOP concepts implemented by the framework, usually compile time and load time weaving frameworks covering a larger subset of AOP concepts. The specific AOP language is important for its expressiveness and easiness of learning. The most successful AOP implementation is AspectJ. It offers compile time or load time weaving and a specific language built as an extension to the Java language.

3. Design Patterns, Aspect Oriented Programming and Metadata

The book *"Design patterns - elements of reusable object oriented software"* [Gamma95] represents the classic work on design patterns. The 23 patterns described in it have the status of software engineering idioms. There is an ongoing research on patterns specific to certain domains, for example Java Enterprise Edition [JEE] patterns [Marinescu02], enterprise architecture patterns [Fowler02] or remoting patterns [Voelter05]. The “Gof” patterns acquired the status of classic patterns due to their generality. They do not belong to a specific domain, but are applicable to generic object oriented design. Due to their popularity, "Gof" patterns were often used to demonstrate the features of a new technology. AOP is not different in this regard. Jan Hannemann and Gregor Kiczales made public the results of using AspectJ to implement the "Gof" patterns in [Hannemann02]. This paper has the same reason behind choosing them.

This chapter is structured in two sections. Section one is an introduction to the design concepts that lead to the "Gof" patterns. Section two presents the 23 "Gof" patterns and the results of applying this paper's approach on them. The patterns are divided as in [Gamma95]. Each pattern section contains its definition, UML diagram and description of its participants. The results are presented for each group of patterns. The patterns showing most improvement are presented in detail in the next chapter.

3.1 Design Patterns

There is a common misconception about design patterns, spread among people newly introduced to them, namely that they are fundamental building blocks of software systems. Design patterns are the embodiment of OOP design principles applied to recurrent software design problems. A system is not a sum of patterns but rather patterns provide help in solving problems in system’s design. The OOP design principles presented in [Gamma95]:

- separation of variance from the invariance
- program to an interface, not an implementation
- use composition over inheritance

Separation of variance from the invariance

There is a big mistake in trying to take into consideration all changes the system has to accommodate. To allow the evolution of the system, one has to create such a design that would facilitate changes. This is accomplished by encapsulating the variance and separating it from the aspects that do not vary. Variance will only cause limited damage when it happens.

Program to an interface, not an implementation

The term interface does not refer only to the interface language construct present in languages like C# or Java, but to the concept behind it. This concept can be expressed as to program to the most general type possible [Olsen07]. This results in a loose coupling of the code, a situation which increases its change resistance.

Use composition over inheritance

Code reuse can be accomplished with two OOP techniques: inheritance and composition. Inheritance is also called "white box" reuse due to the developer needing to know the inner workings of the class to be inherited. Composition is called "black box" reuse as the developer needs only to know the interface of the class. This is one of the most important principles in OOP: favor composition over inheritance. Once a class inherits from another class, there is a strong relationship between them. The problem is more acute in languages which allow only for single inheritance. Using composition, a class is not inheriting another class, but contains a reference to the other class. All operations belonging to the contained class are delegated to it by the class that contains it.

3.2 Aspect Oriented Programming, Metadata and Design Patterns

There are two issues to be remarked in AOP’s criticism: lack of visibility of program flow and difficult debugging; and tight coupling of aspects to the names of language constructs composing the pointcuts, known as "tyranny of the dominant signature" [Laddad05].

A common solution, as shown in [Laddad05], to both issues is to use the metadata facility of the Java platform introduced in version 1.5, namely annotations [Annotations], to mark language constructs to be advised. Annotations are a way to decorate Java language constructs with the purpose of providing information in a declarative manner. AspectJ, starting with version 1.5, offers the possibility of using annotations in the pointcuts. This approach increases the visibility of the program flow and frees the developer from the burden of the "tyranny of dominant

signature". A library of aspects can come with its own set of annotations to be applied on the language constructs to be advised.

This thesis presents an evaluation of AOP implementations of “Gof” patterns, using metadata. "Gof" design patterns implemented using AOP, (AspectJ) have been discussed for the first time in [Hannemann02]. Every pattern was implemented, though only a part of them shown improved characteristics. This is due to the fact that a significant part is using pure OOP techniques without showing crosscutting concerns; AOP's purpose is to encapsulate crosscutting concerns.

The approach used in [Gamma95] is to present each pattern following a certain structure. The sections of the structure are the definition of the pattern, different names for it, the motivation behind, applicability, structure, participants, collaborations, consequences, implementation, sample code, known uses and related patterns. Though there is no clear structure followed in the description of the patterns in this chapter, the focus will be mainly on the definition, motivation, structure and participants sections.

The "GoF" design patterns are divided into three categories:

- Creational Design Patterns
- Behavioral Design Patterns
- Structural Design Patterns

AspectJ and Java annotations limitations

The Java language, starting with version 1.5, accepts annotations on several language constructs, like classes, methods, method arguments, class attributes and variables. The limitation is that annotations on local variables are not accessible in the source, class file or runtime. Hence, AspectJ cannot intercept annotated local variables. This issue will be addressed in [JSR 308].

Advices woven around class constructors in AspectJ have to return an instance of that class or of a subclass. This prohibits the implementation of some patterns, like Proxy.

Evaluation framework

The AOP and metadata implementations are evaluated taking into consideration the following minuses of this approach:

- the usage of the patterns is coupled with an AOP framework;
- the presence of the pattern is hidden by the aspect with negative results (e.g.: the Singleton pattern);

3.2.1 Creational Design Patterns

As their name suggests, creational design patterns provide ways of abstracting the instantiation process resulting in a system which is independent on how objects are created [Gamma95].

a) Abstract Factory

Abstract Factory (see Figure 2) shows how should be modeled the situation in which different families of related objects have to be created.

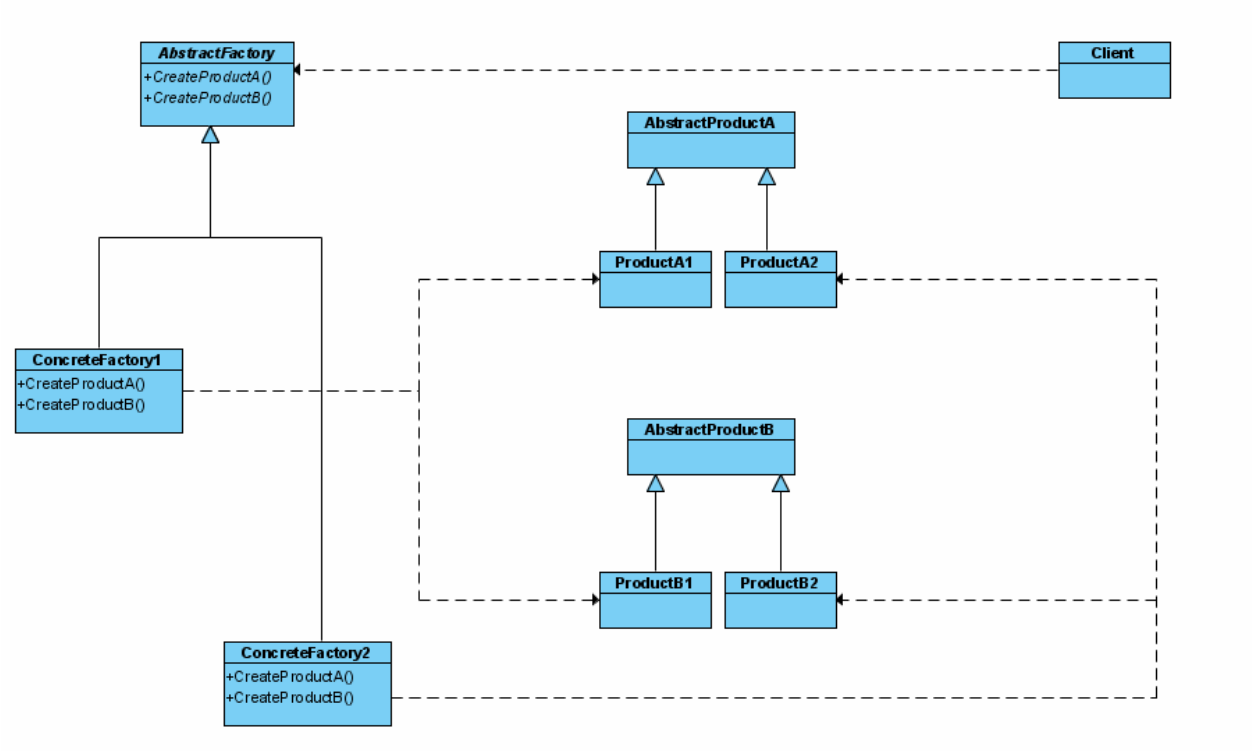

Figure 2: Abstract Factory

AbstractFactory is the common interface used to create objects from a family of objects. There will be an implementation of this interface for each family of objects, in the figure *ConcreteFactory1* and *ConcreteFactory2* corresponding to family 1 and family 2. All the families will have objects of equivalent types, sharing a common interface. Family 1 will have objects of type *ProductA1* and *ProductB1*, while family 2 will have objects of type *ProductA2* and *ProductB2*. *ProductA1* and *ProductA2* share a common interface, namely *AbstractProductA*; the same is valid for *ProductB1*, *ProductB2* and *AbstractProductB*. A good example of this

pattern is the creation of different types of user interface widgets, which have different look and feel but the same functionality.

b) Builder

Builder (see Figure 3) provides a way of separating the creation of a complex object from its representation so that different types could be created by the same construction algorithm.

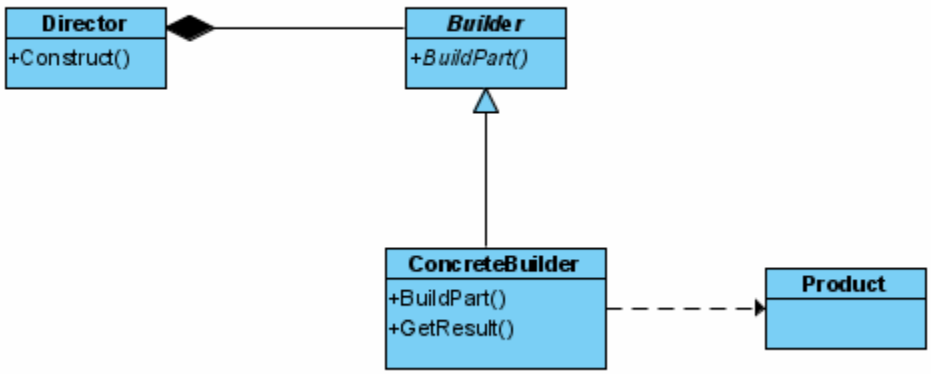

Figure 3: Builder

The *Builder* interface provides methods for creating different parts of the complex object. For creating a certain complex object type, an implementation of the *Builder* interface should be provided. *ConcreteBuilder* is such an implementation. *Director* is the object using a *Builder* implementation to construct the complex object.

c) Factory Method

Factory Method (see Figure 4) defines an interface with an abstract method for creating an object, but defers the object's creation to implementations of that interface.

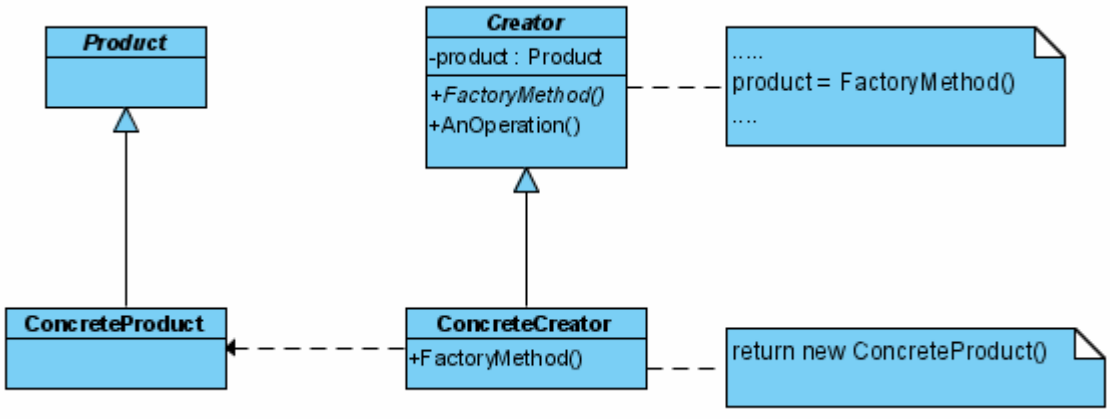

Figure 4: Factory Method

Product is the interface implemented by the objects created with the *FactoryMethod*. *ConcreteProduct* is an implementation of the *Product* interface. *Creator* is the class containing the factory method. The factory method may be used, though not restricted to, inside methods of the *Creator* class to create *Product* implementations, *AnOperation* being an example of such a method. Usually, the factory method is abstract, though *Creator* may contain a default implementation. *ConcreteCreator* is a concrete subclass of *Creator* that provides an implementation of the factory method.

d) Prototype

Prototype (see Figure 5) locates the creation logic of an object inside that object's class. The object is responsible of creating a copy of it.

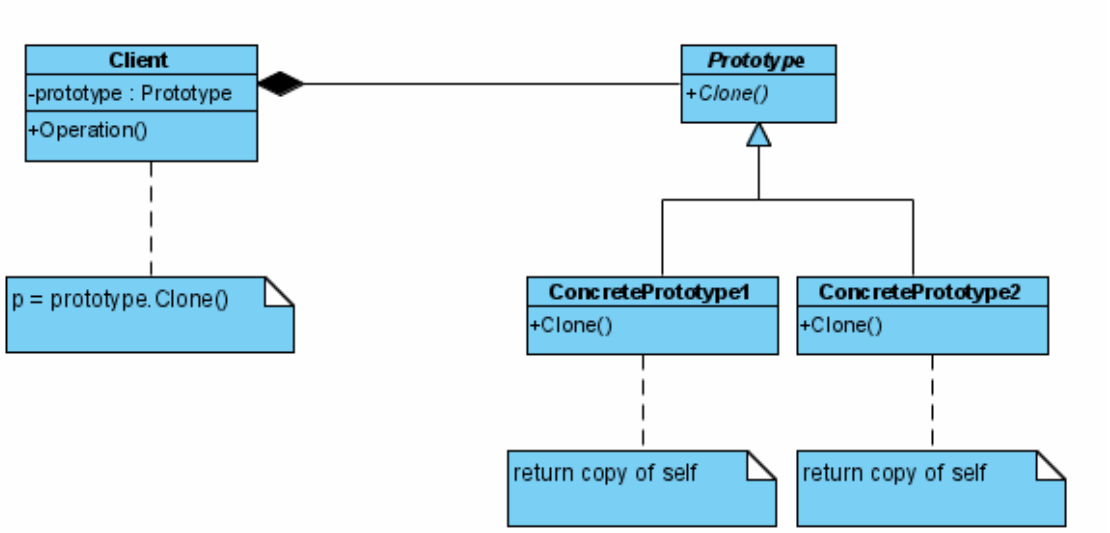

Figure 5: Prototype

Prototype is the interface implemented by objects that are prototypes. *Clone* is the method used for creating a copy of such an object. *ConcretePrototype1* and *ConcretePrototype2* are implementations of the *Prototype* interface. *Client* is the object using *Prototype* implementations in order to create new objects.

e) Singleton

Singleton (see Figure 6) restricts the number of instances of a class to a certain value, providing global access points to those instances.

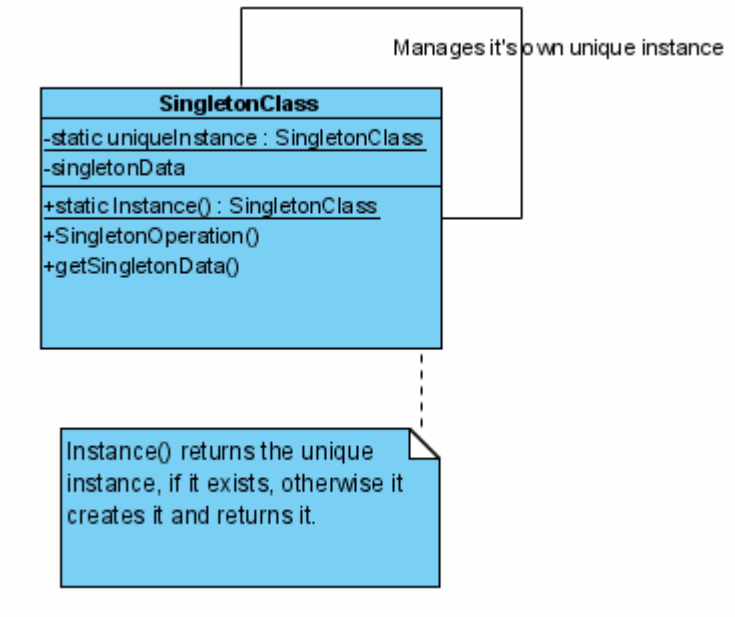

Figure 6: Singleton

The *Singleton* class is an implementation of the *Singleton* pattern. Usually, a *Singleton* has only one instance, but it is not mandatory. It contains a static member of type *SingletonClass*, named *uniqueInstance*, referencing the single instance. The global access point is the static method Instance. The method checks whether an instance has been created, creates one if not, and returns a reference to the instance.

AOP applied to creational patterns

The results of applying AOP to creational design patterns are described as it follows. AbstractFactory, Builder and Factory method make use of metadata and AOP in a similar way. Members that are created using a factory method are annotated as such, with a parameter showing the class that has the factory method to be used. All calls to the constructor of the annotated member are intercepted and the configured factory class is used. In case the factory is changed, only the parameter of the annotation is changed. As for AbstractFactory and Builder, annotations are used to configure what type of factory or builder should be used. Otherwise, static crosscutting is involved in providing a default implementation of the interface, simulating multiple inheritance in Java. For prototype, every class is responsible for cloning itself. There is little benefit from using AOP unless a third party class has to support cloning. In this case static

crosscutting is used for encapsulating the cloning logic. The AOP implementation of the Singleton pattern is presented in the next chapter.

3.2.2 Structural Design Patterns

The patterns in this class define different solutions for how to manage the relationships between the structural parts of the system. These relationships are the result of composing classes and objects in order to form larger structures. There are structural class patterns, using inheritance to compose classes; and structural object patterns, using object composition [Gamma95]. The goal in both cases is to achieve new functionality.

a) Adapter

Adapter (see Figure 7) adapts the interface of a class by converting it to the interface the client expects. Hence, it allows classes with incompatible interfaces to work together.

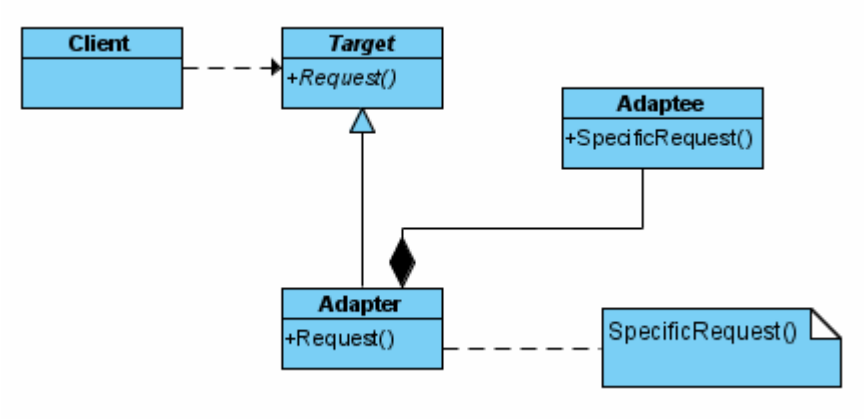

Figure 7: Adapter

Adaptee is the class whose interface has to be adapted. *Target* is the interface the client expects. *Adapter* implements the *Target* interface and contains a reference to an *Adaptee* instance which is used in order to compute the results of the methods in the *Target* interface.

b) Bridge

Bridge (see Figure 8) provides a way to "decouple an abstraction from its implementation so that the two can vary independently" [Gamma95].

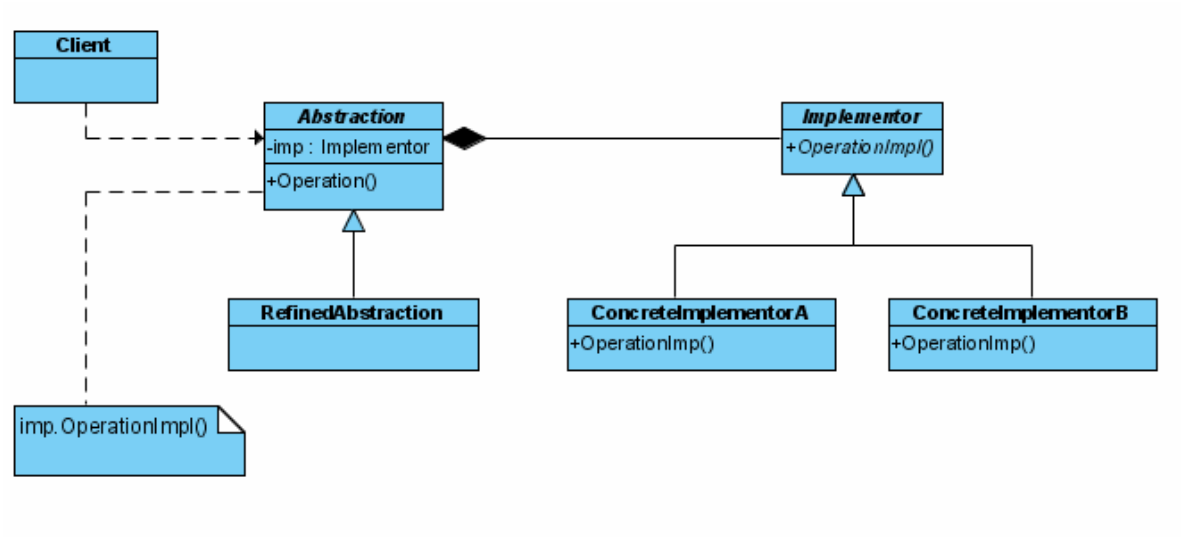

Figure 8: Bridge

There are two interfaces/abstract types involved: *Abstraction* and *Implementor*. Each is the root of an inheritance tree. The composition relation between *Abstraction* and *Implementor* acts as a bridge between the left inheritance tree and the right one. Each subclass of *Abstraction* will have a reference to an *Implementor* instance, but will not be aware of the exact type of the *Implementor* implementation. Hence, they can vary independently.

c) Composite

Composite (see Figure 9) shows how individual objects and compositions of objects can be treated uniformly.

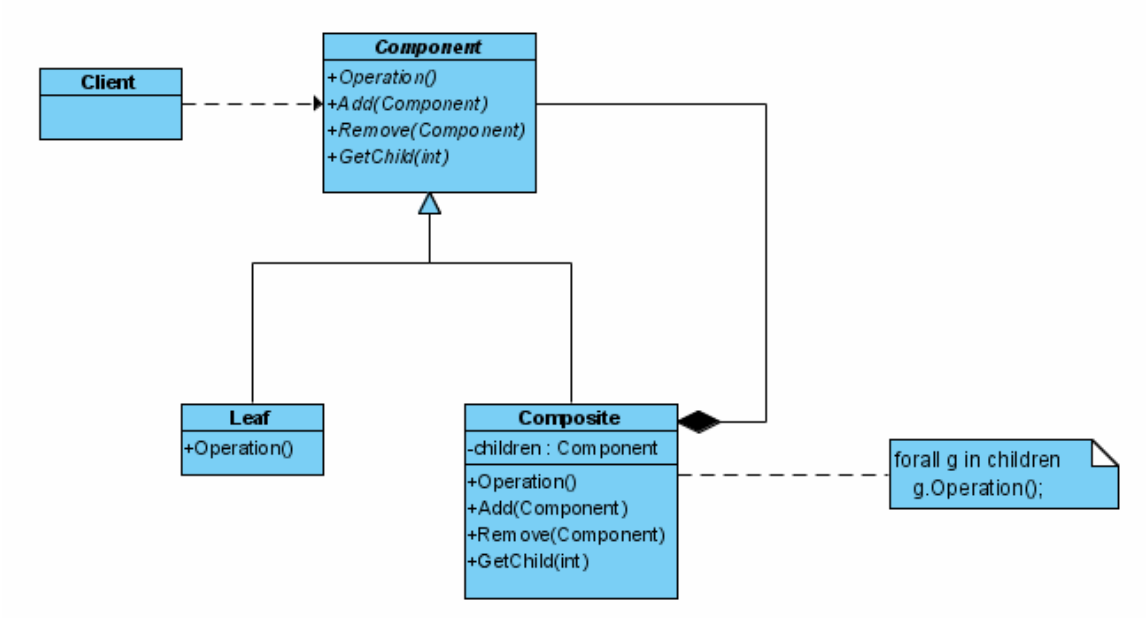

Figure 9: Composite

The *Component* interface acts as a basic type of both individual objects (leaves) and composites of objects. It contains both composite related operations (add, remove) and leaf operations. A leaf class implements only leaf related operations. *Composites* implement leaf and composite related operations.

d) Decorator

Decorator (see Figure 10) dynamically adds functionality to an object without subclassing it.

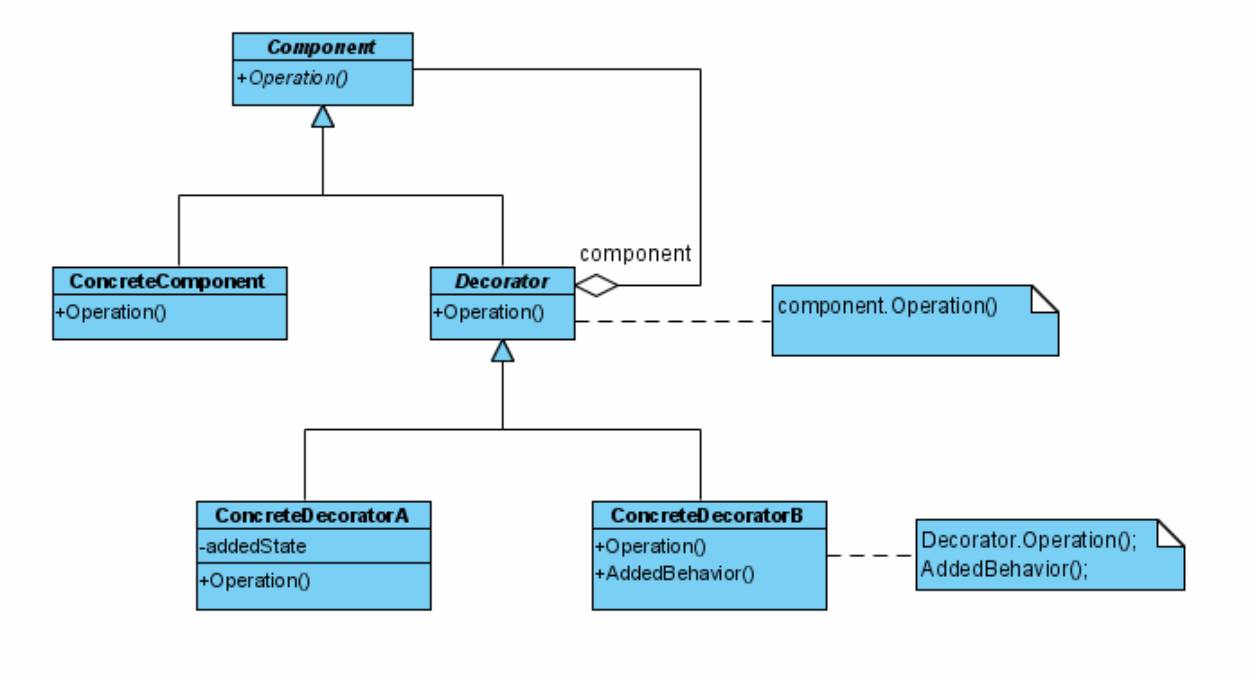

Figure 10: Decorator

Component is the interface shared by both the class to be decorated (*ConcreteComponent*) and decorators. *Decorator* is the interface implemented by concrete decorators. Only operations declared in the *Component* interface can be decorated. Due to sharing a common interface, decorators can also decorate other decorators.

e) Facade

Façade (see Figure 11) provides a single, simplified access point to the interfaces/classes of a subsystem.

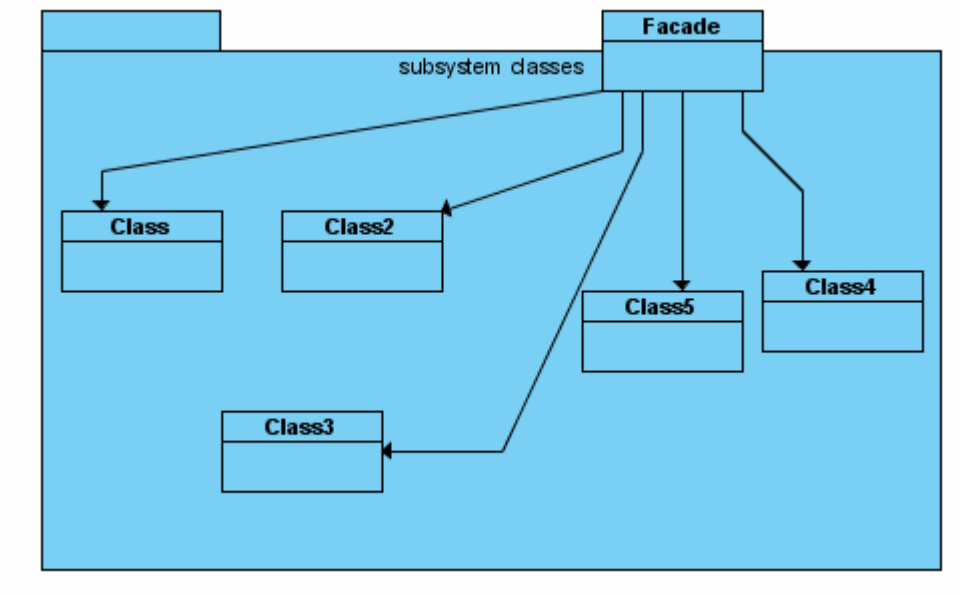

Figure 11: Façade

The *Facade* is the class providing the unified access point. It uses the subsystem's classes to achieve the functionality of the exposed operations. The classes of the subsystem are unaware of the existence of the facade.

f) Proxy

Proxy (see Figure 12) acts as a placeholder for another object, in order to control access to it.

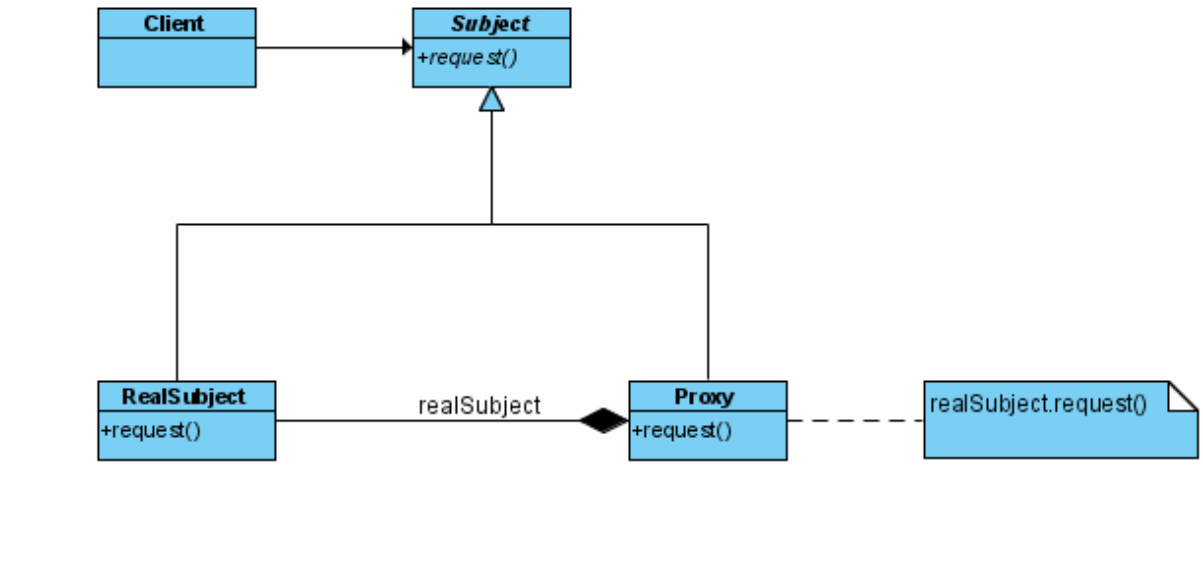

Figure 12: Proxy

Subject is the interface implemented by both the proxy and the object to be "proxied". The client uses this interface, unaware of whether it uses the object or the proxy.

g) Flyweight

Flyweight (see Figure 13) shows how memory occupation can be minimized by sharing as much data as possible between similar objects.

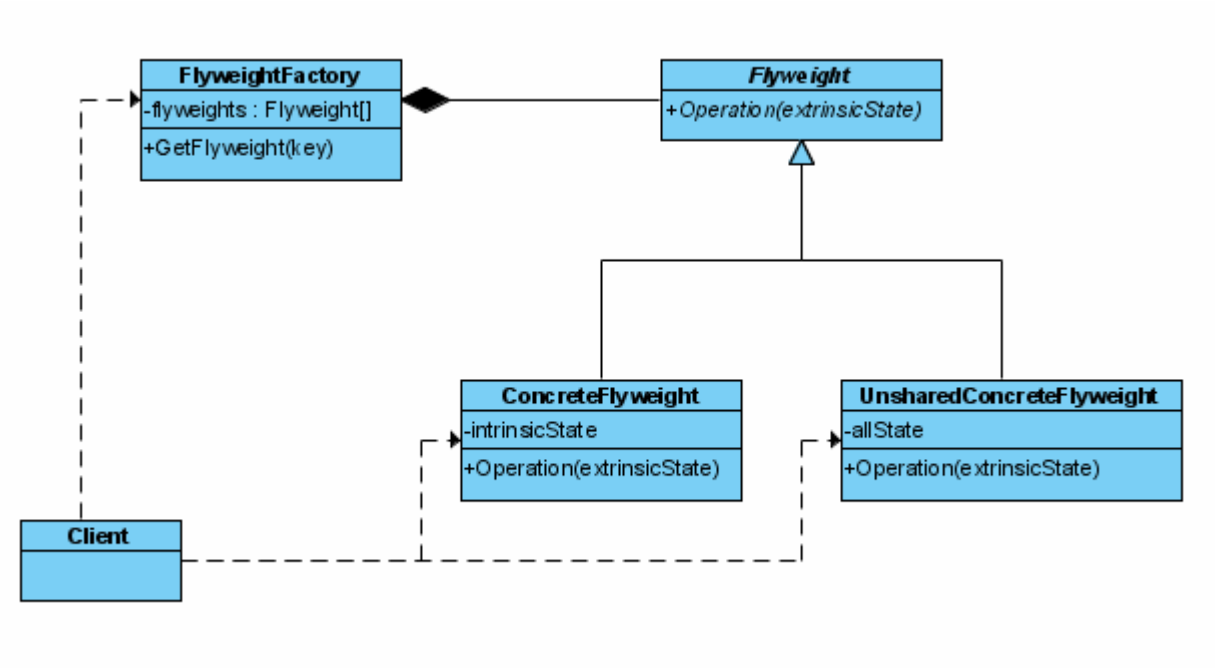

Figure 13: Flyweight

Flyweight is the interface implemented by objects sharing state. The operations defined in the *Flyweight* interface accept as parameters the shared state (extrinsic state). *Flyweight* implementations may have extrinsic state (*ConcreteFlyweight*) or not (*UnsharedConcreteFlyweight*). *FlyweightFactory* is a class responsible with the creation and management of *Flyweight* instances. Already created *Flyweight* instances are stored in a hash structure. If a lookup is performed and there is no entry for that key, a *Flyweight* instance is created, stored in the hash and returned to the client.

AOP applied to structural patterns

Among structural patterns, Facade is the one that cannot be implemented using AOP due to the fact that it presents a generic concept of providing an unified interface to a set of classes. Adapter and Bridge are patterns involving pure OOP techniques. Hence, the AOP version shows little benefit, with the minus of aspect coupling.

In order to avoid the minuses presented in the beginning of the section, the *Flyweight* pattern is improved using AOP by capturing the *Flyweight* creation pointcuts. Thus, plugging/unplugging the pattern resumes to applying / not applying the aspect. Due to the limitations of AspectJ and annotations presented in the beginning of the section, several issues are present in this pattern. The first one is that the class hierarchy has to be designed with the pattern in mind. This means that the concrete heavyweight class has to extend the light flyweight in order to be swapped at instantiation time. The second issue is that the application of the pattern can be configured per flyweight, but at class level. This happens because local variables cannot be annotated. While the second issue will be addressed in [JSR 308], the first one is more severe due to the influence of pattern on class design.

The AOP version of the Composite pattern consists of aspects encapsulating the definition of roles, the structure of children and the logic for managing the children. Though pattern related code is isolated in aspects, performing children management tightly couples the code to the AOP library used.

Due to its purpose, the decorator pattern is a good candidate for a successful AOP and metadata implementation. An aspect is used for every decorated class. Before, after and around advices wrap the concrete component's method calls, in order to add behavior. Annotations are used to mark decorated types. The most serious problem of this approach is the complexity of wrapping decorators in decorators, before wrapping the concrete component. Aspects intercepting aspects and aspect precedence rules can be employed in order to achieve this. The solution is too complex, cannot be performed at run-time and the wrapping takes place at class level. All of these reasons concur to acknowledge the OOP solution as a better implementation of the pattern.

Due to the complexity of the approach, the AOP and metadata implementation of the Proxy pattern is described in the next chapter.

3.2.3 Behavioral Design Patterns

Behavioral patterns deal not only with classes and objects, but also with the communication between them. Hence, they handle complex control flows by shifting the focus from them to how objects are interconnected.

a) Chain of responsibility

Chain of Responsibility's (see Figure 14) purpose is to promote loose coupling between the sender of a request and its receiver. This is achieved chaining objects able to handle the request. Each object has the chance of either handle the request and stop processing, or pass it along the chain.

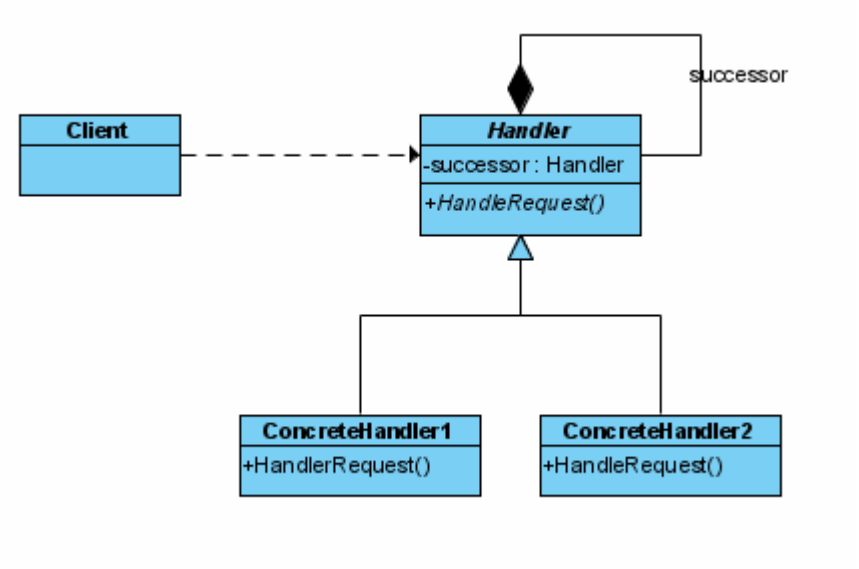

Figure 14: Chain of responsibility

Handler is the common interface of classes implementing request handlers. Each handler has a reference to another handler, used to chain them together. *ConcreteHandler1* and *ConcreteHandler2* are concrete handler implementations. The client gets a reference to the first link of the chain, and calls its *HandleRequest* method.

b) Command

Command (see Figure 15) represents a request as an object by encapsulating it in a class.

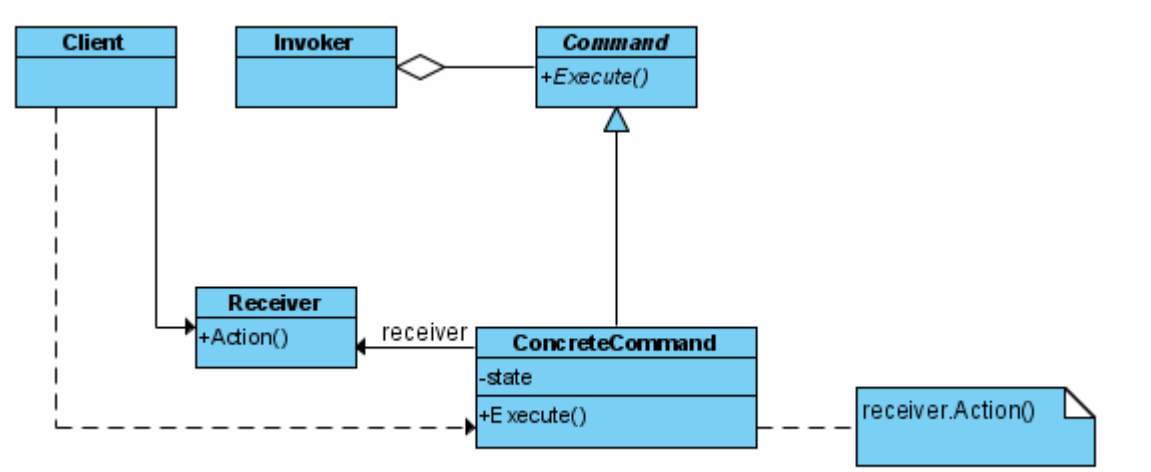

Figure 15: Command

A request is defined by the actions to be performed on its receiver. Hence, an object representation of a request has a reference to its receiver, on which is performing the appropriate actions. Actions are represented by method calls. To provide a unified interface to request objects, the *Command* interface is used. All request objects should implement this interface. *Command* has usually one or two methods (*Execute* and *Undo*) depending whether it is an undo-able command or not. The *Execute* method contains the logic of performing the actions associated with the request on its receiver. *Undo* consists of the operations needed to undo the effect of the actions on the receiver.

c) Interpreter

Interpreter (see Figure 16) presents a solution for representing the grammar of a language and an interpreter to process sentences written in that language, using its representation.

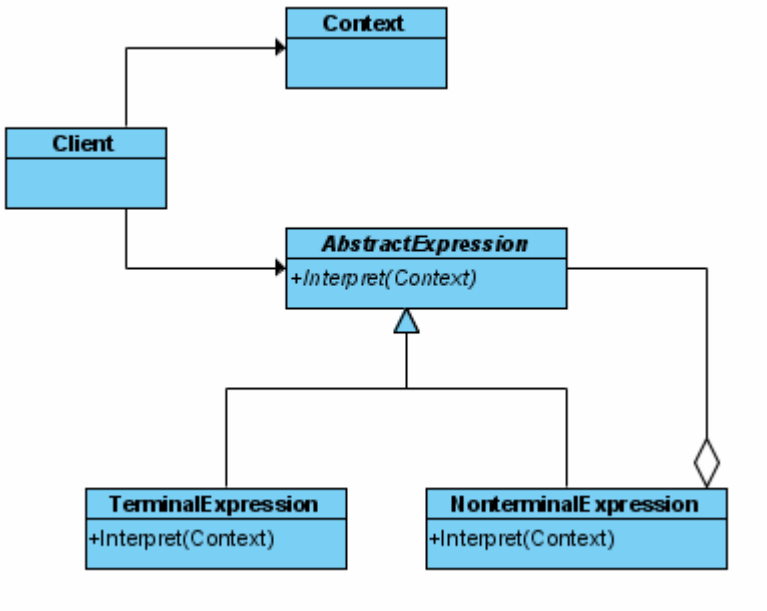

Figure 16: Interpreter

AbstractExpression is the interface implemented by all the nodes of the abstract syntax tree representation of the expression to be evaluated. It defines an *Interpret* method, taking as a parameter a *Context* object, containing information global to the tree (the input string and how much of it has been matched). *TerminalExpression* and *NonterminalExpression* define operations specific to terminal, respectively non terminal symbols in the grammar.

d) Iterator

Iterator (see Figure 17) provides a solution for sequentially accessing the elements of an aggregate object without revealing any details about its implementation.

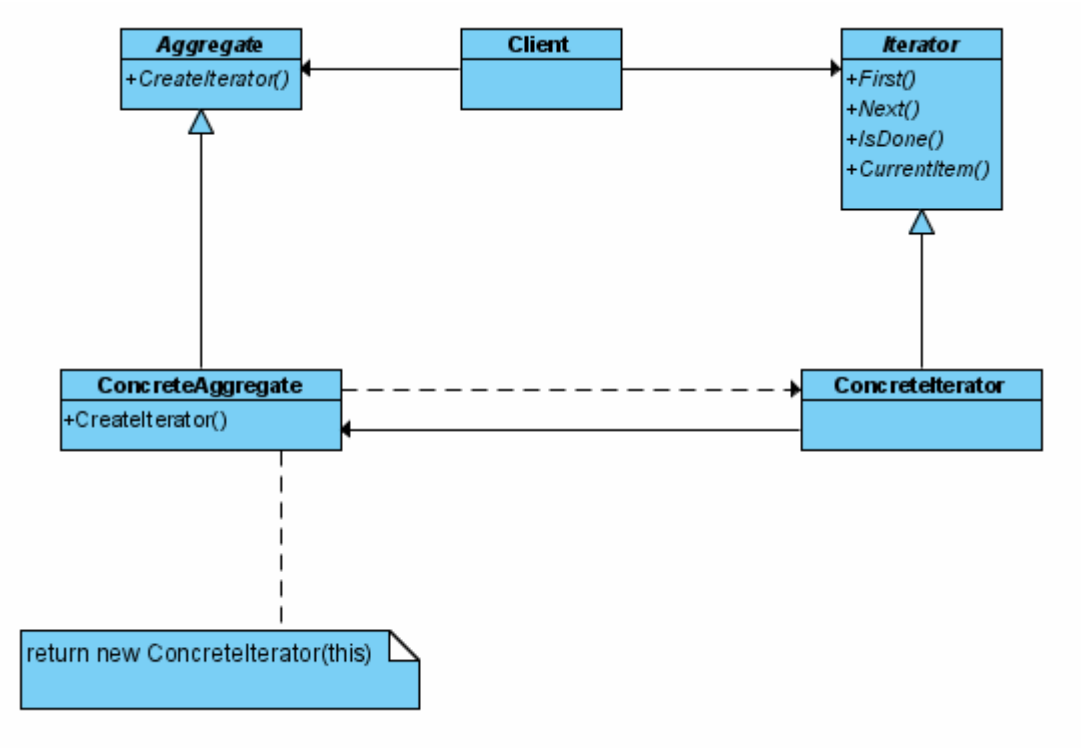

Figure 17: Iterator

Iterator is the interface implemented by iterator objects. It contains methods for traversing and accessing the elements one by one. *Aggregate* is the interface shared by aggregate objects. It contains one method, *CreateIterator*, which returns an iterator object. *ConcreteAggregate* and *ConcreteIterator* are implementations of the aforementioned interfaces. The client uses only the *Aggregator* and *Iterator* interfaces, implementation details remaining hidden.

e) Mediator

Mediator (see Figure 18) allows loose coupling of objects by encapsulating the interactions between them.

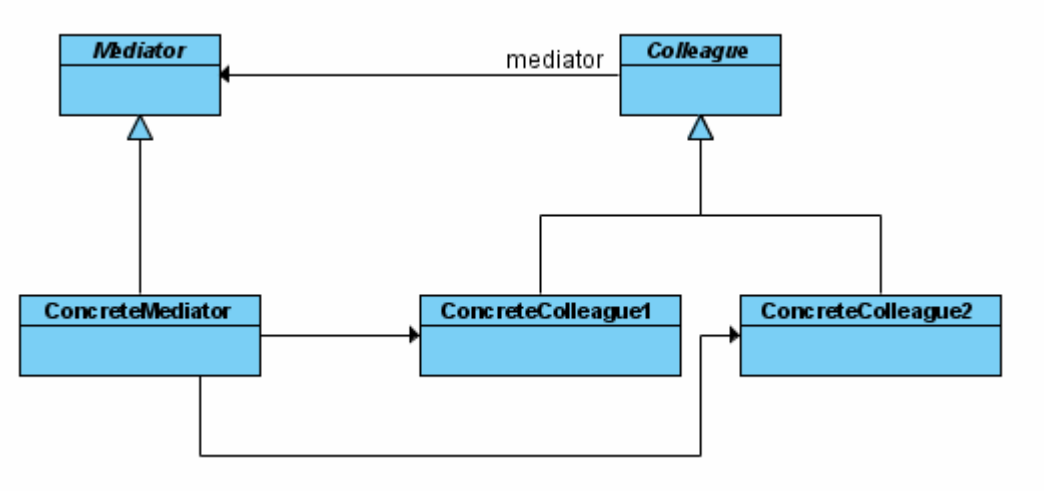

Figure 18: Mediator

Colleague is the interface implemented by the objects that want to be mediated by a mediator. *Mediator* is the interface shared by all mediators. Both interfaces contain the operations needed to enable the communication between mediator and its colleagues. Each colleague knows its mediator, communicating with it when otherwise would communicate with another colleague. *ConcreteMediator* is a concrete implementation of the *Mediator* interface; *ConcreteColleague1* and *ConcreteColleague2* are *Colleague* implementations.

f) Memento

Memento (see Figure 19) captures the internal state of an object without breaking encapsulation. It is used for restoring the state of the object (undo).

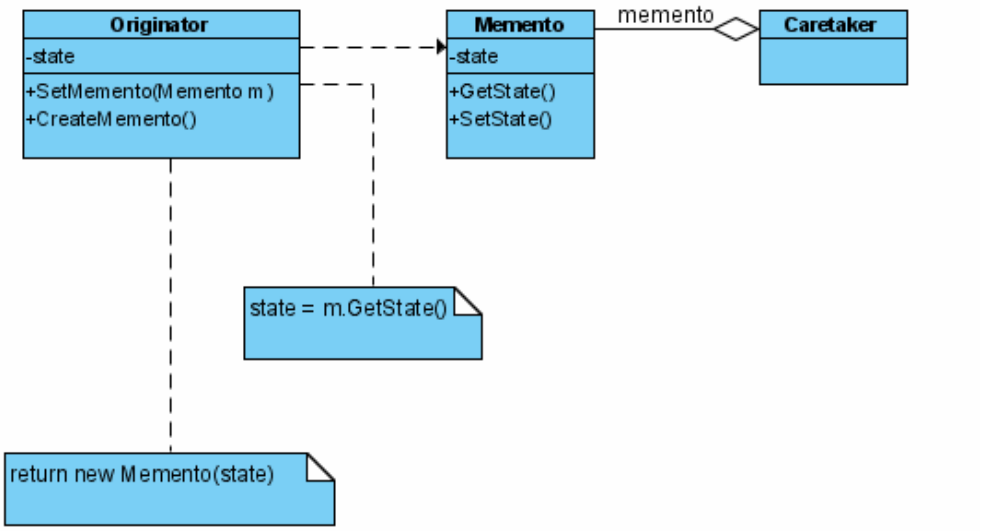

Figure 19: Memento

Originator is the object that can save and restore its inner state using a *Memento*, providing the needed methods for these actions. *Memento* is the object storing the state of the *Originator* object. Its interface consists of methods for setting, respectively getting the inner state of the *Originator*. *Caretaker* is responsible only for safekeeping the memento, without accessing its state.

g) Observer

Observer (see Figure 20) describes a one to many publish/subscribe relationship between objects, one object notifying the others when its state changes.

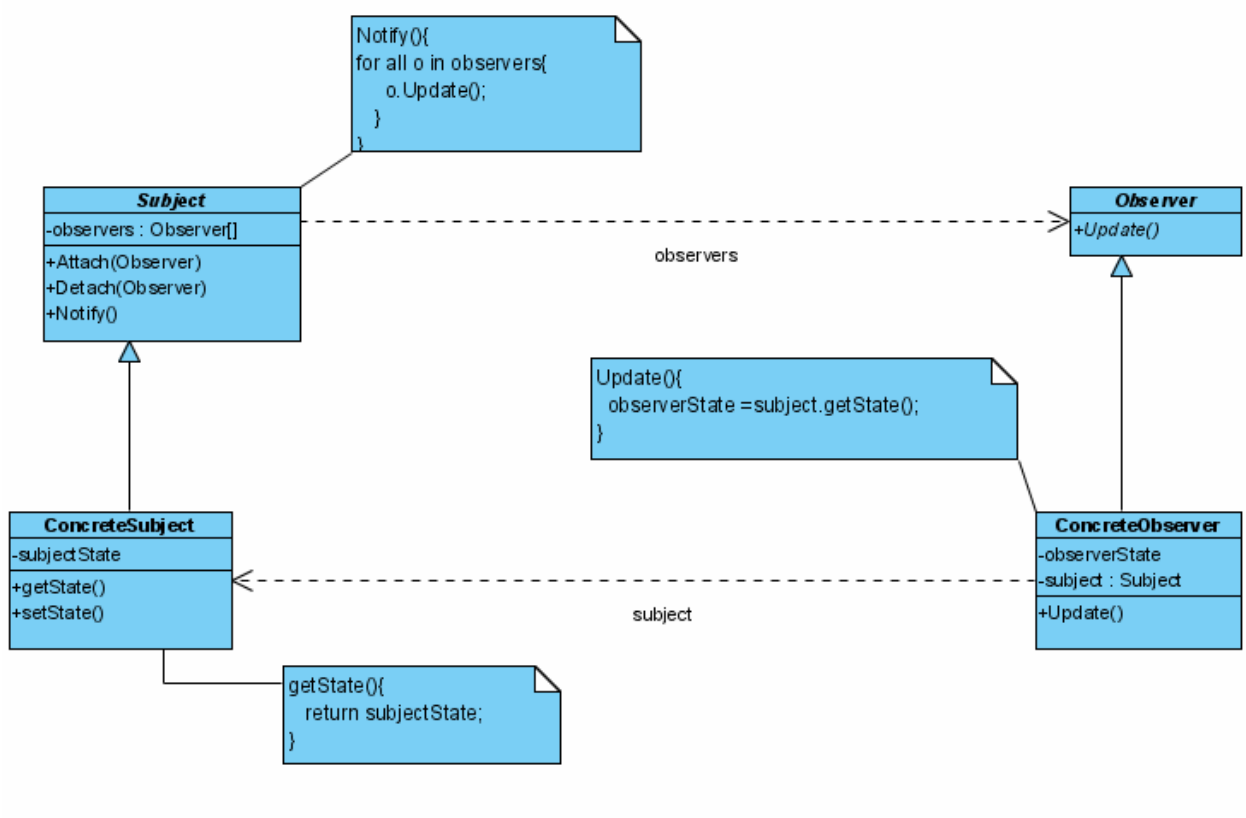

Figure 20: Observer

Observer is the interface implemented by all the objects that subscribe for notifications. It contains an update method, called by the publisher when it changes its state. *Subject* is the interface implemented by the publishers. It contains methods for attaching and detaching subscribers. When notifying its observers, *ConcreteSubject*, a *Subject* implementation, sends itself as a parameter to the update method. Hence *ConcreteObserver*, an *Observer* implementation, uses the *Subject* parameter of its update method to synchronize its state with the new state of the *Subject*.

h) State

State (see Figure 21) shows how an object can change its behavior when its state changes.

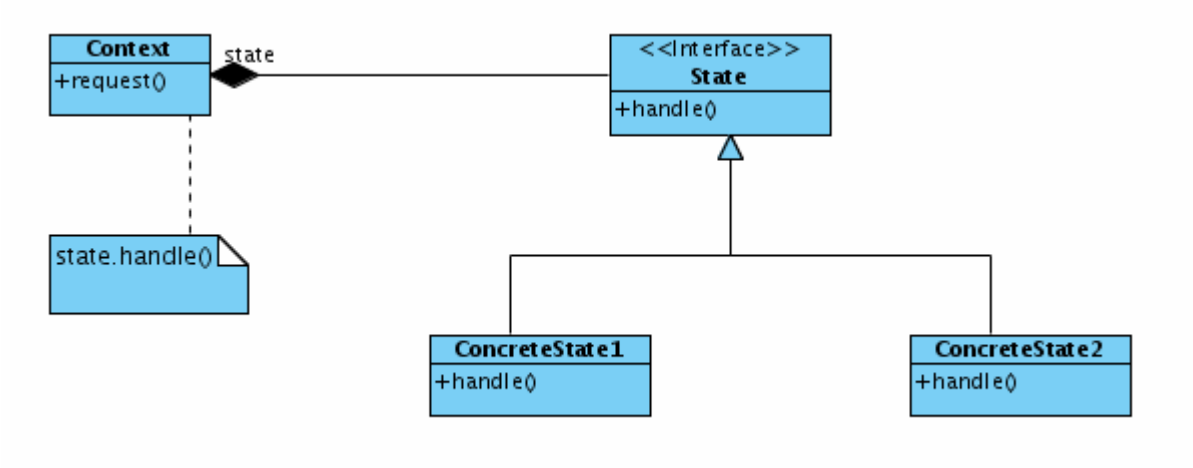

Figure 21: State

Context is the object changing its behavior. This is achieved by encapsulating the behavior in several objects, each defining a state of the *Context* and only one being active at a time. The *Context* delegates to the current state object all the received requests. The *State* interface exposes a set of operations common to all states and is implemented by concrete state objects.

i) Strategy

Strategy (see Figure 22) hides the implementation details of a set of related algorithms behind a common interface. The client will only be exposed to this interface; hence algorithms can vary independently of it.

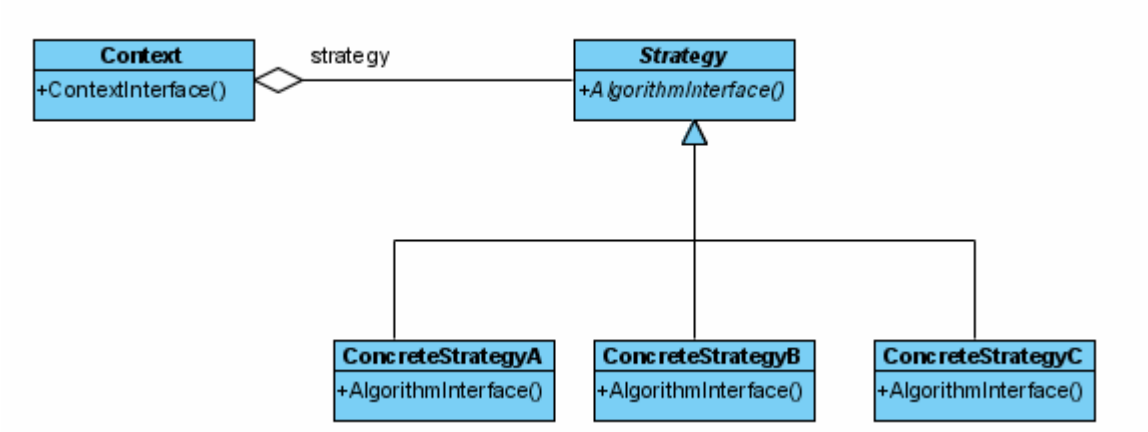

Figure 22: Strategy

Strategy is the interface common to the set of related algorithms. *ConcreteStrategyA*, *ConcreteStrategyB* and *ConcreteStrategyC* are different related algorithms, encapsulated in classes implementing the *Strategy* interface. *Context* is the object using one of the related

algorithms in order to perform a task. For this, it has a reference to a *Strategy* object, to which it delegates all algorithm related responsibilities. The *Context* is configured with the concrete algorithm to be used, but is unaware of its concrete type. *Context* can also provide an interface for the algorithms to access its data.

j) Template method

Template method (see Figure 23) defines the steps of an algorithm as abstract methods in an abstract class in order to allow subclasses define them. The skeleton of the algorithm is implemented in the template method of the abstract class as calls of the abstract methods and can not be changed in subclasses.

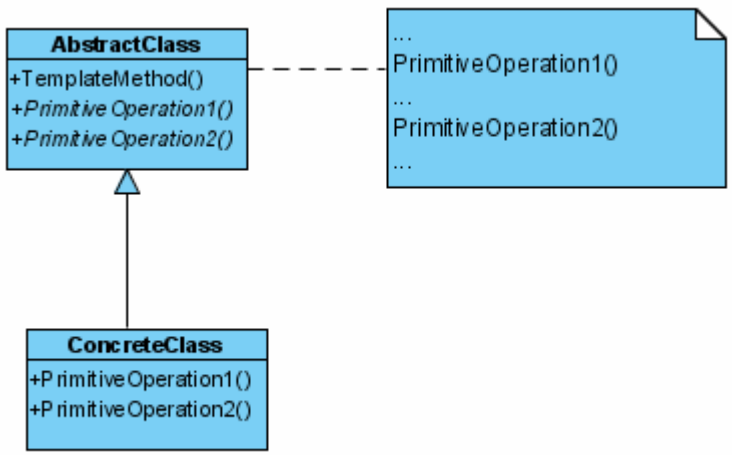

Figure 23: Template Method

AbstractClass is defining abstract primitive operations of the algorithm as abstract methods. It also defines the skeleton of the algorithm in a template method. The template method is calling primitive operations, as well as other methods of the *AbstractClass* or other objects. *ConcreteClass* subclasses *AbstractClass* and provides a specific implementation for the primitive operations.

k) Visitor

Visitor (see Figure 24) encapsulates the operations to be performed on the elements of a structure of objects to allow the addition of new operations without any change to the classes of the elements.

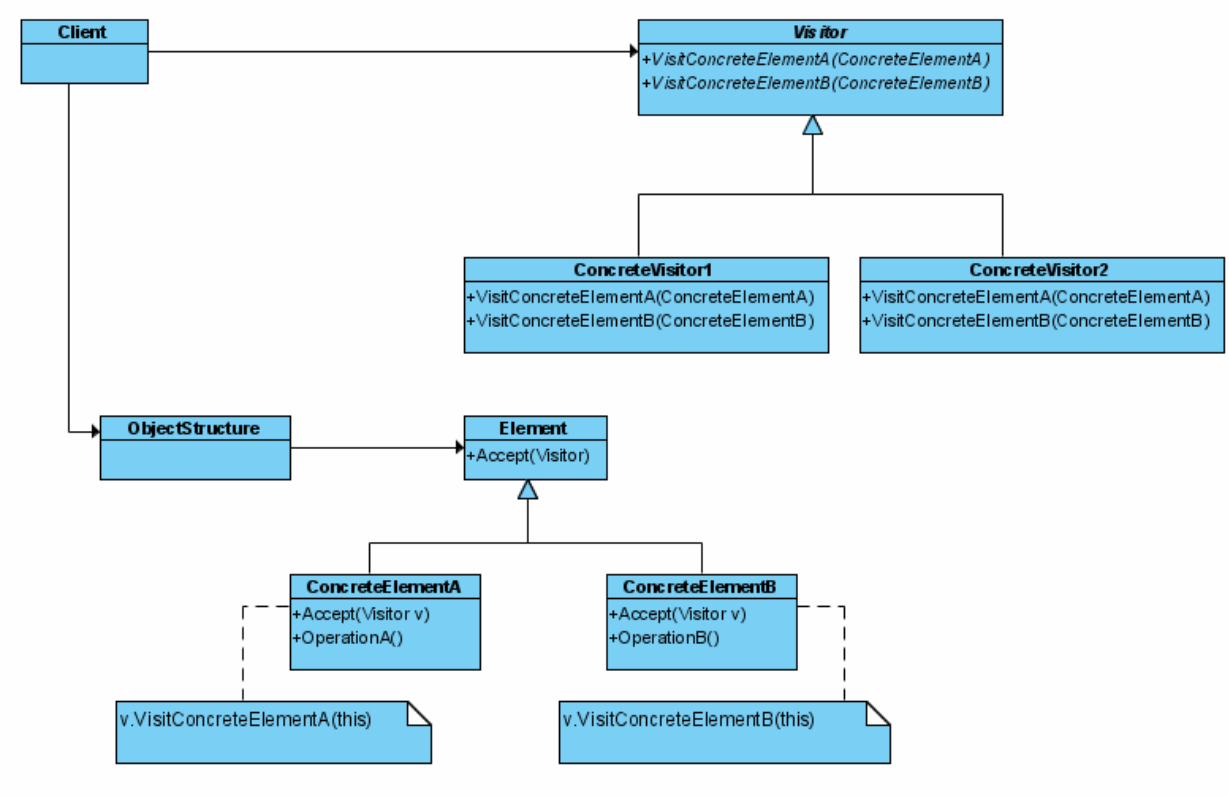

Figure 24: Visitor

Visitor is the interface implemented by all objects representing operations on the elements of the object structure. It contains one method for the type of each element in the structure. Every method accepts as a parameter an object of the class it deals with. Each concrete visitor has to provide an implementation for all the methods, consisting of how the operation it represents is performed on the specific class. Concrete visitors also provide the context for the algorithm and store the accumulated results as local state.

Element is the interface common to all elements of the object structure. It contains one method, *Accept*, expecting a parameter of type *Visitor*. This method consists of a call to the *Visitor* method specific to the class of the element implementing the *Accept* method.

The Interpreter pattern is a generic solution for the interpretation and representation of sentences written in a user defined language. Due to its nature, like in Facade's case, AOP and metadata cannot be employed for improving the OOP implementation.

The Template Method and the Strategy pattern involve only pure OOP techniques to achieve their purpose. They exhibit no crosscutting concerns to be encapsulated by using aspects. In case of the Strategy pattern, there are two approaches to use metadata and AspectJ. One is to use annotations to configure the algorithm to be used. This is something generic, not restricted to the Strategy pattern. The second approach is common the both the Strategy and the Template Method patterns. AspectJ is to employ static crosscutting in order to provide a default implementation for the methods of the Strategy interface or of the Template Method's *AbstractClass* as a workaround for Java's single inheritance.

The Iterator, Chain of Responsibility, Visitor, Command, Memento and Mediator patterns have similar AOP implementations. Pattern related code is encapsulated in aspects, making use of static crosscutting to inject it in the participants. This approach is also beneficial in code reusability, as it provides default implementations of interface methods. The drawback is that the code initializing the pattern (iterator creation, relations between mediator and colleagues) is tightly coupled to the AOP framework used. The exception is the Visitor pattern, where the code is coupled with the aspect's name. The general AOP approach for these patterns is:

- define interfaces for the pattern's roles in an abstract aspect;
- define data structure to manage the relationships between participants in the abstract aspect (only if needed);
- assign roles to participants in a concrete sub-aspect.

These steps can be identified with the Director AOP design pattern, presented in [Miles04].

It is problematic to use metadata to express the code currently coupled with the AOP framework due to the fact that annotations can only express compile time relations. This means that runtime instances cannot be put in relation by using annotations. As an example, a mediator and its colleagues can be configured using annotations, but only at their declaration. It is similar to the problem of the Decorator pattern, which benefits a lot from run time wrapping of instances, unavailable in the AOP implementation.

The AOP and metadata implementation of the Observer and State patterns is presented in the next chapter.

3.3 Summary of results

The 23 "Gof" patterns can be classified as follows according to how they are implemented using AOP and metadata:

The AOP and metadata approach exhibits limited or no benefit	*The AOP and metdata approach shows benefits*
Façade	Visitor
Interpreter	Composite
Adapter	Chain of Responsibility
Strategy	Proxy
Decorator	Factory
Iterator	Flyweight
Bridge	Singleton
Abstract Factory	Observer
Builder	State
Command	
Prototype	
Memento	

Table 1: Classification of design patterns

3.4 Conclusions

Design patterns were widely introduced to software engineering by [Gamma95]. The most important achievements of this work are establishing a common vocabulary for software engineers and presenting the design principles behind the patterns. Due to their ubiquitousness, generality and popularity, the 23 "Gof" patterns were often used to prove the viability of a new technology. This was done by implementing them in that technology and analyzing the result. AspectJ was used to implement the "Gof" pattern as a proof of AOP's possibilities. AOP's critics include the "tyranny of the dominant signature" and hiding of program flow. Annotation mixed with AspectJ come as a solution to overcome those critics. The biggest obstacle in the path of a good AOP implementation of a pattern is its pure object oriented nature and the degree of generality. The best implementations correspond to patterns that exhibit the pattern related behavior as a crosscutting concern to the functionality of the objects involved in the pattern. Good examples are the following: Singleton, State, Proxy, and Observer.

4. Aspect Oriented Programming and Metadata Implementation of Design Patterns

The four design patterns that display significant improvement are: Singleton, Observer, State, and Proxy. They have in common the fact that the pattern related code crosscuts the code specific to the participants in the pattern. All four implementations make use of AOP and annotations in a similar manner:

- Annotations mark and configure the participants in the pattern.
- Aspects contain pointcuts capturing joinpoints defined by annotations and encapsulate the logic of the pattern.

This approach results in the following improvements:

- The pattern is plugged or unplugged depending on the presence or absence of annotations.
- The coupling between the participants' types and the pointcuts is based only on the annotations, resulting loose coupling.
- Loose coupling of the codebase and the AOP framework used. The presence of an AOP framework is not mandatory. Annotation processing tools can be involved to interpret the annotations and generate the pattern's code.
- Annotations improve the view of the program's flow.
- Pattern related code is isolated in aspects, not interfering with the logic encapsulated in the participants in the pattern.

4.1 Singleton

Description

The Singleton design pattern (see Figure 25), as it is described in [Gamma95], represent a way in which a class can have only one instance per application also providing a global access point to that instance.

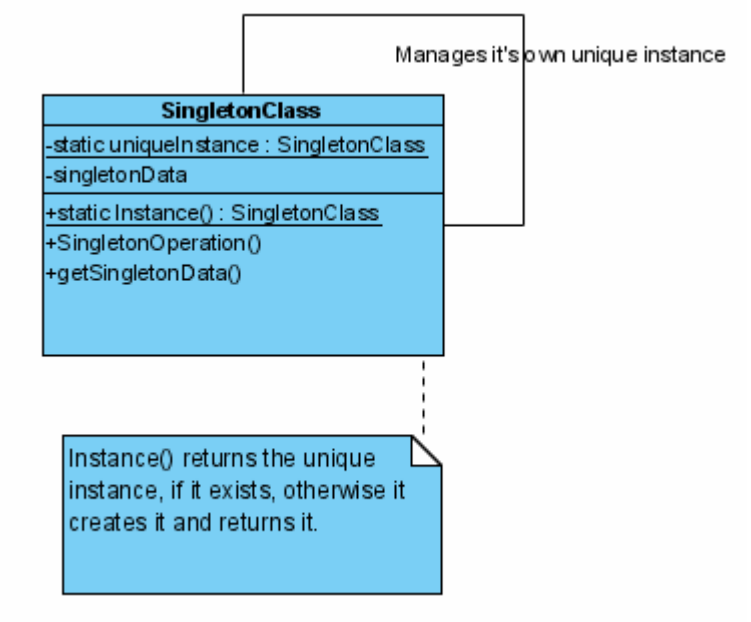

Figure 25: Singleton UML

Implementation details

A Singleton pattern implementation has to take into consideration three aspects. One is how the Singleton instance is retrieved: using a method (usually named *Instance*() or *getInstance*()) or the normal way to create objects (using the new keyword). Second aspect consists of defining the behavior of the Singleton when it is extended by a subclass. Finally, the third aspect is the real uniqueness of the Singleton. This last aspect is usually met in distributed applications.

.New() or .Instance()

The OOP approach for this pattern needs an *Instance*() method for getting an instance of the Singleton class. This approach is invasive because it requires modification to a class to make it a Singleton. In the same time, the developer using this class will know that it is a Singleton. When considering the AOP approach, a decision has to be made whether the Instance() method will be added to classes that are supposed to be Singletons, and thus allowing for the same code as in a OOP approach, or if the constructor call will be intercepted, hence hiding from the developer if the class is a Singleton or not. By providing marker interfaces or annotations, this information can still be available, even though it will not be as straightforward as an Instance() call.

Sub classing

The decision of what should happen when the Singleton class is sub classed is a decision to be taken by the developer. If the Java platform is to be considered, several options are available, like overriding the instance method to return an instance of one of the subclasses of the Singleton class or to declare the Singleton class "final" so it can not be extended.

Uniqueness

This problem appears usually in distributed applications. A discussion of these issues is done in [Fox01]. A singleton class is unique per class loader or virtual machine, so multiple Java Virtual Machines generate multiple instances of the Singleton. The developer faces two alternatives: accept this situation and design and use the pattern with these aspects in mind, or manage this situation in the Singleton creation logic. One solution for the latter case would be a central Singleton registry. The problem is that the coherence of the singleton has also to be handled also. It is very important for these issues to be acknowledged when the design of the singletons in an application is taking place.

Aspect Oriented Implementation

The crosscutting nature of this pattern is the creation of the object. The OOP approach requires a protected or private constructor and a public static Instance method to create the objects. In the Instance method is encapsulated the logic for creating the Singleton. Due to these constructs, the Singleton class is not a POJO (Plain Old Java Object) [Fowler00] and the pattern is invasive. AOP offers a clean solution for encapsulating the invasive nature of this pattern. This pattern is commonly used in dependency injection frameworks [Fowler04] as for example the Spring Framework which provides the Singleton mode as the default instantiation model. Briefly

explained, a dependency injection framework handles the instantiation of the classes registered with it and of the dependencies of these classes on other classes. The classes' registration, instantiation policy and dependencies are usually described in a declarative way, such as an external XML configuration file or annotations. Hence, no modification is required to the class' code to register it with the framework. When a client requires an instance from the framework, it receives a fully initialized object with all its dependencies resolved. The drawback is all the objects have to be created using the framework's Abstract Factory [Gamma95] implementation; as a result, the application's code is highly coupled with the framework. In the case of containers, the client's code usually performs a lookup of the instance that it needs, and the container handles the rest. An AOP implementation of the Singleton pattern is useful in the development of a framework or a container but also in an application's development to avoid the coupling of its code to a framework for the Singleton pattern. Of course, an OOP implementation can be used any time, with the cost that to make a class a Singleton, its structure has to be modified.

Marking the Singleton: @Annotation or Interface

Several options are available to mark a class as being a Singleton:

- Hardcode the class to be handled as a Singleton - a poor choice as this implementation is inflexible to changes.
- Use a marker Interface (e.g. *Singleton*), with no methods, like the *Serializable* Interface is used (available in Java pre 1.5).
- Use an annotation (e.g. *@Singleton*) that can be applied to a Class.
- Use an abstract Aspect to implement the Singleton creation logic, with an abstract pointcut that defines the classes to be handled as Singletons. This aspect has to be extended by a concrete aspect that should provide a definition of the abstract pointcut.

The first option is limited due to being inflexible. As for the next two, it depends on the Java Runtime Environment (JRE) where the application will run. If it's a pre 1.5 JRE (annotations not supported), then a marker interface seems like the only solution of those two. If the JRE is 1.5 or later, both can be used however an annotation makes more sense because of the following two reasons:

- Annotations are meant to express metadata [JSE1.5].
- The Singleton nature of a class can be regarded as metadata.

As for the last options, it depends a lot on how much AOP is to be included in the application's development. If aspects are first class citizens in the development an approach as in [Hannemann02] could be a good as it isolates all the concerns of the pattern in an abstract aspect that is extended to provide a concrete implementation. For the time being, AOP is still in an adoption phase in software development and developers search for incremental ways of including AOP in their work, without coupling to it but gaining value from using it. For fulfilling these expectations, AOP should play more like a gluing role, an orchestrating role, rather than an intrusive one.

Source Code & Sequence Diagram

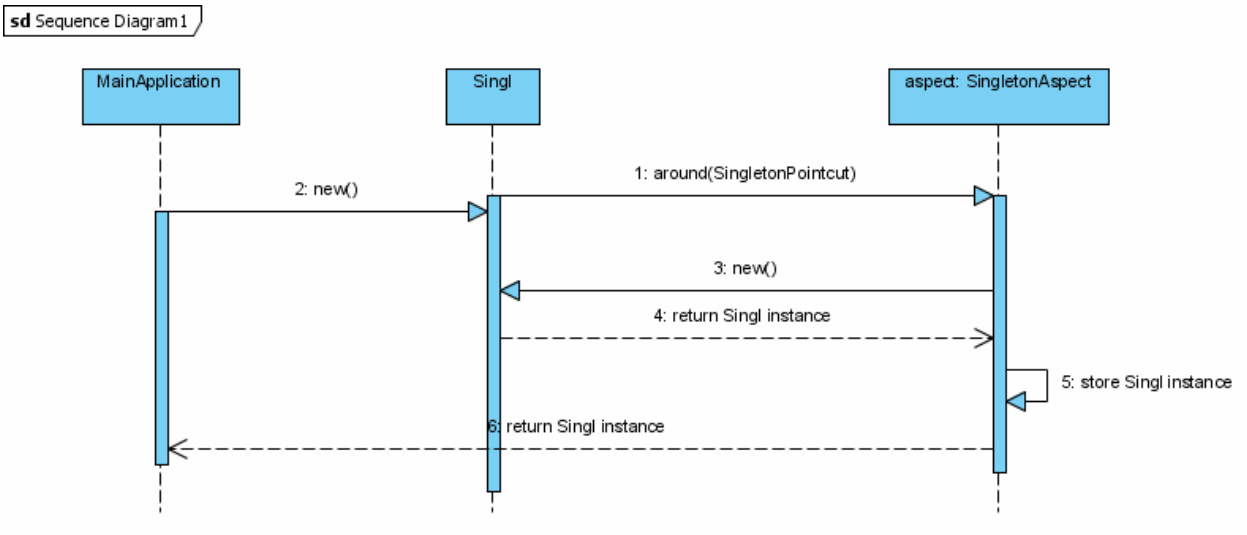

Figure 26: Singleton sequence diagram

The Singleton annotation is used to mark classes as singletons. It may contain an attribute to express whether the access to the singletons should be synchronized or not. In case annotations cannot be used a marker interface *Singleton* will replace it to mark classes as singletons. For synchronized singletons, another marker interface *SyncSingleton* that extends the *Singleton* interface is used. To apply these interfaces, only the *AnnIntrAspect* and the *SinglAspect* have to be modified to accommodate changes. Instead of marking classes with annotations, the interfaces will be used. The overridden abstract *pointcut* has also to be changed so that it intercepts the call to the constructors of the subtypes of two interfaces.

Singleton.java

```
package jns.sing.ann;

/*
 * Annotation for marking classes as Singletons.
 */
public @interface Singleton {

}
```

Three aspects are involved: *AnnIntrAspect*, *SingletonAspect* and *SinglAspect*. *AnnIntrAspect* is applied before the *SingletonAspect* and *SinglAspect*. This aspect deals with marking types as Singleton and should be used for classes that are not under the control of the developer, such as third party library classes. *SingletonAspect* is an abstract aspect that contains the abstract pointcut *SingletonPointcut*, on which the singleton creation logic is applied. This logic is encapsulated in an *around* advice, applied around the constructor call of the *Singleton* classes. Also, this aspect contains a *WeakHashMap* to contain the instances of the singletons. In case a instance does not exist, it is created by calling *proceed* in the around advice and storing the return value in the singletons *Map*. *SinglAspect* extends the abstract *SingletonAspect* and overrides the abstract pointcut to specify what constructors to be intercepted.

AnnIntrAspect.aj

```
package jns.sing.as;
import jns.sing.ann.*;

/*
 * Aspect marking classes as Singletons using the @Singleton annotation.
 * Useful when the source code is not available (third-party libraries) or
 * there is a need of not coupling classes with the annotations.
 */
public aspect AnnIntrAspect {

/*
 * Declares this Aspect to be applied before any other aspect in the system.
 * Has to be like this so that the classes are marked as Singletons before the
 * SingletonAspect is applied.
 */
    declare precedence : AnnIntrAspect, * ;

/*
 * Modifying classes to be annotated with @Singleton annotation.
 */
    declare @type : jns.sing.dp.Singl  : @Singleton ;
    declare @type : jns.sing.dp.Singl2  : @Singleton ;
    declare @type : jns.sing.dp.Singl12  : @Singleton ;
}
```

SingletonAspect.aj

```
package jns.sing.as;


import java.util.WeakHashMap;

/*
 * Abstract aspect performing the singleton creation logic.
 * Contains an abstract pointcut ( SingletonPointcut()) that is used in
 * intercepting the creation of Singleton classes.
 *
 */


public abstract aspect SingletonAspect {

/*
 * Used as a registry of Singleton. It registers the singletons in the system
 * having as key the hash code of their class names and as value their unique instance.
 * Empty in the beginning, it grows as singleton instances are created.
 */
    private WeakHashMap singletons = new WeakHashMap();


/*
 * To use this aspect, it has to be extended by providing a proper definition of this
 * abstract pointcut. This pointcut captures the calls to the constructors of the
classes
 * that should be singletons. It's abstract in order to provide flexibility in marking
 * classes as Singleton and to not be coupled with the Singleton marking option.
 */
    public abstract pointcut SingletonPointcut();

/*
 * Advice implementing the singleton creation logic. It is applied around
 * the SingletonPointcut(). If an instance of the class whose constructor had been
 * called exists in the singleton registry, the constructor called is bypassed and that
 * instance is returned. Otherwise, the constructor is called, the returned instance is
 * stored in the Singleton registry and after that returned to the client.
 */

    Object around() :
        SingletonPointcut() {
        Object tmp = null;

        int key = thisJoinPoint.getSignature().getDeclaringType().hashCode();
        tmp = singletons.get(key);
        if(tmp == null){
             tmp = proceed();
            singletons.put(key, tmp);
        }
        return tmp;
    }
}

/*
 * Aspect that extends SingletonAspect, providing an expression for
 * the abstract pointcut. It defines the SingletonPointcut to capture
 * the calls to the constructor of the classes annotated with the
 * @Singleton annotation.
 */
```

SinglAspect.aj

```
package jns.sing.as;

import jns.sing.ann.*;

public aspect SinglAspect extends SingletonAspect{
```

```
    public pointcut SingletonPointcut() :
        call ((@Singleton *).new(..));


}

/*
 *  Main application that creates several instances of classes marked as singletons.
 */
```

Main.java

```
package jns.sing.prg;

import jns.sing.dp.Singl;
import jns.sing.dp.Singl12;
import jns.sing.dp.Singl2;

public class Main {

    public static void main(String[] args) {

        Singl t1 = new Singl();
        Singl t2 = new Singl();

        Singl2 t3 = new Singl2(10);
        Singl2 t4 = new Singl2(20);

             }

}
```

Conclusions

The Singleton pattern is a perfect candidate for an AOP implementation. The code specific to the pattern is a crosscutting concern in relation to the logic implemented by the class that is a singleton. An aspect is used to wrap around an object's creation (constructor call) and provide the Singleton related behavior. Annotations make a perfect mechanism to mark classes as singletons. Using AOP and metadata, the pattern can be plugged or unplugged without any side effect. The only criticism of the proposed approach is the fact that the client is not aware whether it is using a singleton or not. This issue is addressed in the OOP implementation by using an static method *Instance()* to retrieve the singleton's instance instead of the "*new*" keyword.

4.2 Observer

Description

The Observer design pattern (see Figure 27) shows how a one-to-many relationship between objects can be represented so when the object on the *one* side of the relationship changes its state, the objects on the *many* side of the relationship are notified [Gamma95].

Two roles are present in this pattern:

- The object on the *one* side of the relationship is called the Subject.
- The objects on the *many* side of the relationship are called Observers**.**

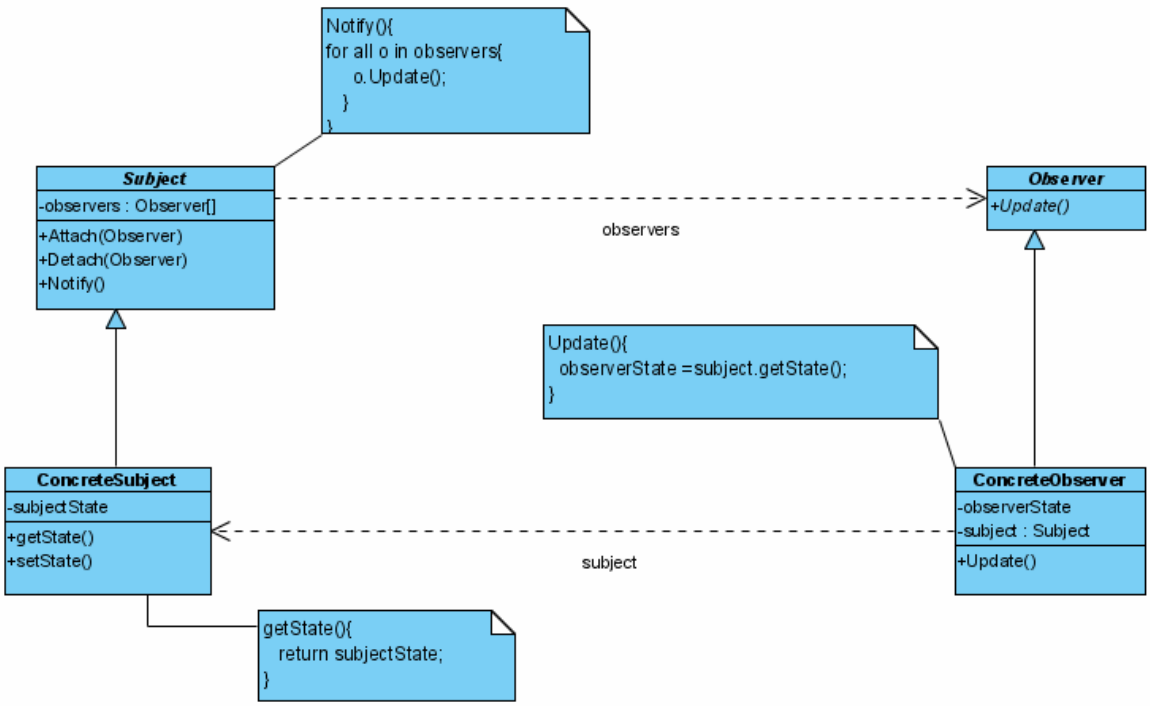

Figure 27: Observer UML

Implementation details

To implement this pattern in an OOP way is to have two interfaces corresponding to the two roles. If one class is supposed to play any of the roles, it should implement the specific interface. Roles are interfaces in object oriented languages that do not support multiple inheritance. An abstract class can be used instead of an interface, but that will not allow the Subject to inherit from another class, thus not an elegant choice. The Java Standard Edition provides a default

implementation of this pattern. There is an *Observable* class in the *java.util* package, playing the role of the *Subject* interface, which has to be extended in order to create a particular *Subject* class. Also, an *Observer* interface is present in the same package, which has to be implemented to allow a class to play the *Observer* role.

One feature is a default implementation for methods of an interface. Some languages, like Ruby [Ruby], allow this using the concept of mixins, although Java does not. Nevertheless, using AOP's static crosscutting, a default implementation for a method of an interface can be provided. Static crosscutting is how this pattern is implemented in [Hannemann02].

Aspect Oriented Implementation

The approaches used for this pattern both in [Hannemann02] and [Miles04] are based on assigning roles to objects using Interfaces and providing default implementations for some of the methods of the interfaces. While using interfaces is the only possible approach in a pre JRE 1.5 environment, the presence of annotations [Annotations] in JRE implementations starting with JRE 1.5 allow other possibilities to express the roles and dependencies of the objects. It should be mentioned that everything expressed with annotations or role interfaces can be also represented as an external configuration file, be it XML, Java properties or any other format. Going into this way, the result will be more framework-like than a part of the language, adding the complexity of maintenance of the XML files.

In the implementation of the Observer pattern presented in the following, an annotation based model is chosen. The general requirements for the participants in this pattern are listed below:

- Each Subject has to have a collection of Observers.
- The Subject should allow Observers to register/deregister themselves to it. The notification will be sent to all registered Observers.

In [Hannemann02] and [Miles04] all the data structures, methods and interfaces related to the pattern are included in the aspects. While everything is kept together, the code of the pattern is tight coupled to AspectJ. Due to the goal to assure a loose coupling, in the case presented here, these elements are separated from the aspects in a different package.

There are several places where the relations between a Subject and its Observers can be kept:

- In the aspect itself, as it is presented in [Hannemann02] and [Miles04].

- In the Subject itself, using static cross-cutting.
- Behind a public interface that offers options to manage them.

In the case presented here, the third option has been chosen, reasons being in the goal of separating the involved classes as much as possible from the pattern implementation. In this case, the code managing the relations between objects is totally unaware of aspects or pattern related code other than the objects involved.

Source code and Sequence Diagram

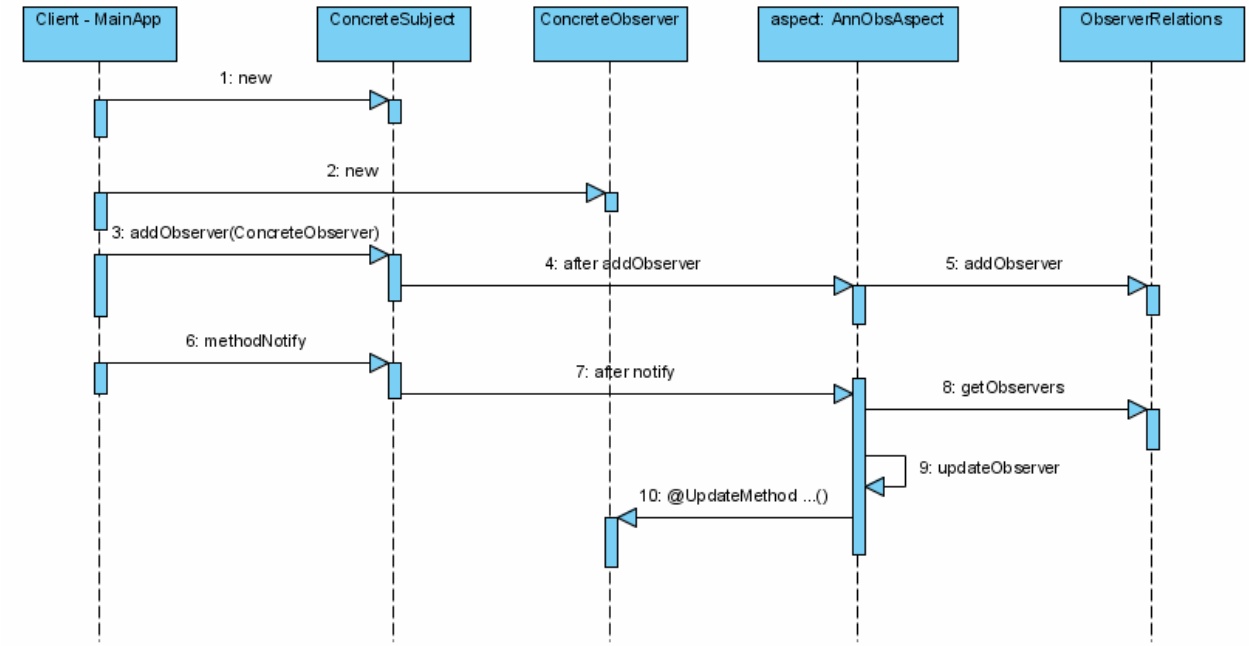

Figure 28: Observer Sequence Diagram

The source code will be presented classified in regard to whether it belongs to the Subject-Observers relationships, Subject role, Observer role and the aspects that provide the gluing.

Subject - Observers relationships

ObserverRelationships.java

```
package jns.observer.dp;

import java.util.LinkedList;
import java.util.WeakHashMap;

/*
 * Interface for managing the relations between Subjects
```

```
 * and Observers.
 */
public class ObserverRelations {

  /*
   * Data structure for handling the relations between
   * Observers and Subjects.
   * The key of the hash map is the hash code of the Subject.
   * Each Subject has a linked list of Observers.
   */
  private static WeakHashMap<Integer, LinkedList<Object>> subjToObs =
                        new WeakHashMap<Integer,LinkedList<Object>>();

  /*
   * Method for adding an Observer to a Subject
   */
  public static void addObserver(Object subject,Object observer){
    LinkedList<Object> observers = subjToObs.get(subject.hashCode());
    if(observers == null){
      observers = new LinkedList<Object>();
      subjToObs.put(subject.hashCode(), observers);
    }
    observers.add(observer);
  }

  /*
   * Method for getting the Observers of a Subject.
   */
  public static LinkedList<Object> getObservers(Object subject){
    LinkedList<Object> observers = subjToObs.get(subject.hashCode());
    return observers;
  }
}
```

This class provides an interface for managing the relations between the Subjects and the Observers. The data structures and the methods are static, providing a single point for managing the relations per application. This class is a good candidate to apply the Singleton pattern [Gamma95]. The mapping between a Subject and its observers is done using a *HashMap* whose key is the hash code of the Subject object and the value is a *LinkedList* which contains the Observers for that particular Subject.

Subject

There are several aspects to be discusses here. The Subject class is marked using several annotations. It has two methods to add and to remove Observers, methods having an empty body and one parameter of type *Object*, representing the Observer to be added or removed. These methods are annotated with *@AddObserver* and *@RemoveObserver* annotations. After those methods are called, an aspect performs the adding/removing of the observer sent as parameter. It does not have to be the only parameter; several strategies could be employed here, like annotating the parameter to be added as Observer. Also, in this case, the relations between Subjects and Observers could be hidden inside the aspect as it is the only one aware of it. It was kept out to make possible the addition if a custom management of the relationships. The method of the

Subject that triggers the notifications of the Observers should be annotated with the *@NotifyAfter*. The notifications take place after the method is executed. If the use scenario requires, an empty method can be used.

Subject.java

```
package jns.observer.ann;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

/*
 * Annotation used to mark a class as playing
 * the Subject role.
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface Subject {

}
```

This annotation is used for declaring a class as playing the subject role.

AddObserver.java

```
package jns.observer.ann;

/*
 * Annotation used to mark a method as the method whose parameter
 * will be used to be added as Observer
 */
public @interface AddObserver {

}
```

RemoveObserver.java

```
package jns.observer.ann;


/*
 * Annotation used to mark a method as the method whose parameter
 * will be used to be removed as Observer
 */
public @interface RemoveObserver {

}
```

Those annotations are used for marking methods of a *@Subject* annotated class as the methods whose parameter will be the Observer to be added or removed.

ConcreteSubject1.java

```
package jns.observer.dp;

import jns.observer.ann.AddObserver;
import jns.observer.ann.NotifyAfter;
import jns.observer.ann.RemoveObserver;
```

```
import jns.observer.ann.Subject;

/*
 * Example implementation of a Subject class
 */
@Subject
public class ConcreteSubject1 {

  /*
   * empty methods annotated to trigger the adding
   * and removing of observers
   */
  @AddObserver
  public void addObs(Object observer){}

  @RemoveObserver
  public void removeObs(Object observer){}

  /*
   * method that will trigger the notifications of
   * the observers
   */
  @NotifyAfter
  public void methodNotify(){
    System.out.println("my method");
  }

}
```

A class marked as being the Subject.

Observer

To mark a class as an Observer the @Observer annotation has to be used. Also, the method called when the Observer is notified has to be a method which takes one parameter of type Object and has to be annotated with the @UpdateMethod annotation. The same discussion as for the parameters of the methods annotated with @AddObserver and @RemoveObserver is also valid here.

Observer.java

```
package jns.observer.ann;

/*
 * Annotation used to mark a class as playing
 * the Observer role.
 */
public @interface Observer {

}
```

The annotation used to mark a class as playing the Observer role.

UpdateMethod.java

```
package jns.observer.ann;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

/*
 * Annotation used to mark the method to be called when the Observer
 * is notified.
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface UpdateMethod {

}
```

The annotation used to mark the method of a class annotated with *@Observer* as the method to be called when the Observer is notified by its Subject.

ObserverImpl1.java

```
package jns.observer.dp;

import jns.observer.ann.UpdateMethod;

/*
 * Example implementation of an Observer
 */
public class ObserverImp1  {

  /*
   * The Notify method, takes as parameter the subject
   */
  @UpdateMethod
  public void update(Object subject) {
    // TODO Auto-generated method stub
    System.out.println("observer 1 "+subject.toString());
  }

}
```

ObserverImpl2.java

```
package jns.observer.dp;

import jns.observer.ann.UpdateMethod;

/*
 * Example implementation of an Observer
 */
public class ObserverImpl2  {

  /*
   * The Notify method, takes as parameter the subject
   */
  @UpdateMethod
  public void update(Object subject) {
    // TODO Auto-generated method stub
    System.out.println("observer 2 " + subject.toString());
  }

}
```

ObserverImpl1 and ObserverImpl2 are classes marked as being Observers.

Aspects

There is an abstract base aspect *ObserverAspect*. It provides the logic for notifying the Observers of a Subject and two abstract extension points: an abstract pointcut *ObsNotifyPointcut* and an abstract method *updateObserver* that performs the invocation of the update method on the Observers of a Subject. The logic for notifying the Observers is encapsulated in an *after* advice that is applied after the *ObsNotifyPointcut*. It consists of getting the Observers for the Subject, whose method triggered the *pointcut*, and calling the abstract method *updateObserver* on each Observer. Due to these design decisions, there is no coupling between the base aspect and the way the Subjects and Observers are marked. *AnnObsAspect*, a concrete aspect, extends the base aspect and provides an implementation for the two extension points. The two extension points can be summed up as when the notification should be triggered - the pointcut; and who should handle the notification - the abstract method.

ObserverAspect.aj

```
package jns.observer.as;

import java.util.LinkedList;
import jns.observer.dp.*;

/*
 * Aspect used to handle the notification of the Observers
 * when the trigger method is called on the Subject.
 */
public abstract aspect  ObserverAspect {


  /*
   * Pointcut to define the point in the flow of the program
   * that will trigger the notifications of the Observers.
   */
  public abstract pointcut ObsNotifyPointcut(Object subject);

  /*
   * After advice that will handle the notification of the Observers
   * after the trigger point.
   */
  after(Object subject) : ObsNotifyPointcut(subject) {

    System.out.println("after notify method");
    LinkedList<Object> observers=
                              ObserverRelations.getObservers(subject);

    for(Object o : observers){
      updateObserver(subject,o);
```

```
    }
  }

  /*
   * Abstract method that will handle the invocation of the Update
   * method on the observers.
   * It is abstract to allow the customization of the marking of this
   * method.
   */
  protected abstract void updateObserver(Object subject,
                                                    Object observer);
}
```

AnnObsAspect.aj

```
package jns.observer.as;

import jns.observer.dp.*;
import java.lang.annotation.Annotation;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

import jns.observer.ann.*;


/*
 * Concrete aspect that extends the ObserverAspect, uses Subjects and
 * Observers marked with annotations. Also, provides a way to add and
 * remove Observers based also on annotations.
 */
public aspect  AnnObsAspect extends ObserverAspect {

  /*
   * Intercepts calls to methods annotated with @AddObserver which
   * belong to classes annotated with @Subject.
   */
  public pointcut AddObserverPointcut(Object subject,Object observer) :
            call ( @AddObserver * (@Subject *).*(Object))
                  && target(subject) && args(observer);

  /*
   * After advice, adding the observer after the AddObserverPointcut
   */
  after(Object subject,Object observer) :
                                AddObserverPointcut(subject,observer){

    ObserverRelations.addObserver(subject, observer);
  }

  /*
   * Definition of the Notify pointcut.
   */
  public pointcut ObsNotifyPointcut(Object subject) :
      call ( @NotifyAfter * (@Subject *).*()) && target(subject);

  /*
   * Implementation of the updateObserver method.
   * In this case, it invokes the method
   * annotated with the @UpdateMethod annotation.
   */
  protected void updateObserver(Object subject, Object observer) {

    Method[] methods = observer.getClass().getMethods();
    for (Method met : methods) {
      for (Annotation an : met.getAnnotations()) {
        if (an instanceof UpdateMethod) {

          try {
```

```
            met.invoke(observer, subject);
          } catch (IllegalArgumentException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
          } catch (IllegalAccessException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
          } catch (InvocationTargetException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
          }
        }
      }
    }
  }

}
```

The Main Program

Main.java

```
package jns.observer.prg;

import jns.observer.dp.ConcreteSubject1;
import jns.observer.dp.ObserverImp1;
import jns.observer.dp.ObserverImpl2;
import jns.observer.dp.ObserverRelations;

public class Main {

  /**
   * @param args
   */
  public static void main(String[] args) {
    // TODO Auto-generated method stub
    ConcreteSubject1 subj = new ConcreteSubject1();

    subj.addObs(new ObserverImp1());

    subj.addObs(new ObserverImpl2());
    subj.methodNotify();


  }

}
```

Conclusions

The crosscutting concern present in the Observer pattern is the notification of observers by the subject. To address this situation an aspect is used to trigger the notification of the observers subscribed to a subject after the call of the *Notify* method. In the approach presented in this thesis, the aspect also handles the management of observers. Annotations are used to configure the participants in the pattern. The *Notify* and *Update* methods are marked with annotations. For adding/removing observers, empty annotated methods are provided. This is the same solution as

the one used by the annotation version of AspectJ 5. The drawback of this approach is that the Subject class must be aware of playing a role in an Observer pattern.

4.3 State

Description

The State pattern (see Figure 29) presents a solution to the situation in which an object should change its behavior when it's internal state changes, appearing to be changing its class.

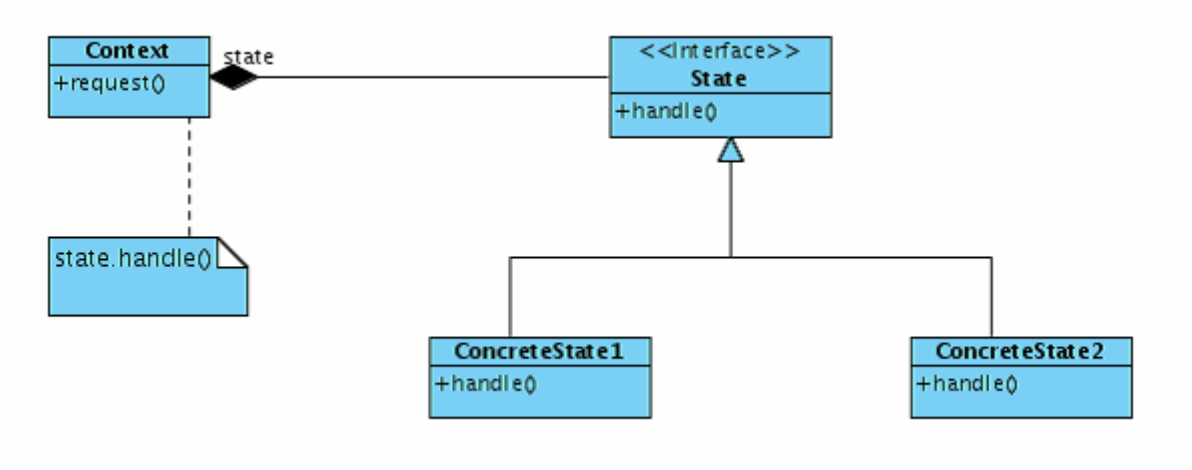

Figure 29: State UML

Implementation details

This pattern assigns the following roles to the participants: *Context* and *State*. The *Context* is the role of the class that changes its behavior. *State* is the interface to the internal state of the *Context*. Concrete states are concrete implementations of the *State* interface, providing the behavior of specific states. The *Context* class has a reference to a *State* instance, to which it forwards all behavior related requests. Though there are several *ConcreteState* implementations available, only one is available at a time for an instance of the *Context*. The crosscutting concerns of the State pattern are *ConcreteStates* type specification and instantiation policy, and *State* transitions.

Aspect Oriented implementation

ConcreteStates type specification and instantiation policies

Usually the types of the *ConcreteStates* to be used with a *State* pattern are hard coded in the logic of instantiation. Also, there are several policies that could be used for instantiating the *ConcreteStates*. All the required instances can be created when the *Context* instance is created, or

they could just be instantiated when needed for the first time. Choosing one policy over the other is a particular decision for every case in which the pattern is used.

As a solution, a *State* annotation has been developed, which has two attributes: states, an array of *Class* objects and *instantiationPolicy*, a *StateInstantiationPolicy* value object. *StateInstantiationPolicy* is an enumeration containing the types of instantiation policies, in this case *EAGER* (all *ConcreteStates* are created when the *Context* object is created), or *LAZY* (a *ConcreteState* is created when it is needed for the first time). The states attribute contains a list of the types (*Class* objects) of the *ConcreteStates*. Because a *Context* needs and initial state, the first element of the states list is used as the first one. An aspect will intercept the construction execution of the *Context* class, read the attributes of the annotation and create the instances of the *ConcreteStates*, if *EAGER* policy is chosen.

State management data structure

There are different ways in which the instances of the *Context* could be associated with an instance of the *Context*. In this example, the aspect manages a *Map*, having as key the *Context* object and as values maps having as key *Class* objects (the types of the *ConcreteStates*) and as values the instances of the *ConcreteStates*. In case of a *LAZY* initialization, only the necessary entries will be created in the managing data structure, the instances of the *ConcreteStates* being added as needed. Another solution would be to use static crosscutting to include the list of *ConcreteStates* instances in the *Context* class.

State Transitions

As a method is called on the state attribute, if it is needed, the state has to be changed to pointing to another *ConcreteState* implementation. Not all method calls trigger a state transition, but some do. For this, a *StateTransition* annotation has been developed, having as attribute of type class, *nextState*. This attribute is applied on methods and will indicate the type of the *ConcreteState* used after the method call. Depending on the instantiation policy, an instance will be fetched from the state management data structure or created if it does not exist.

As mentioned in Chapter 3, the Java platform currently does not support local variable annotation. When this issue is addressed, states and contexts can be configured locally, not at class level. The AOP approach is the same, only the creation pointcut has to be changed.

The State pattern's best example is a *TCPConnection* class as the Context, a *TCPState* interface and several *TCPState* implementations. This example was used as the AOP implementation of the *State* pattern.

Source Code & Sequence Diagram

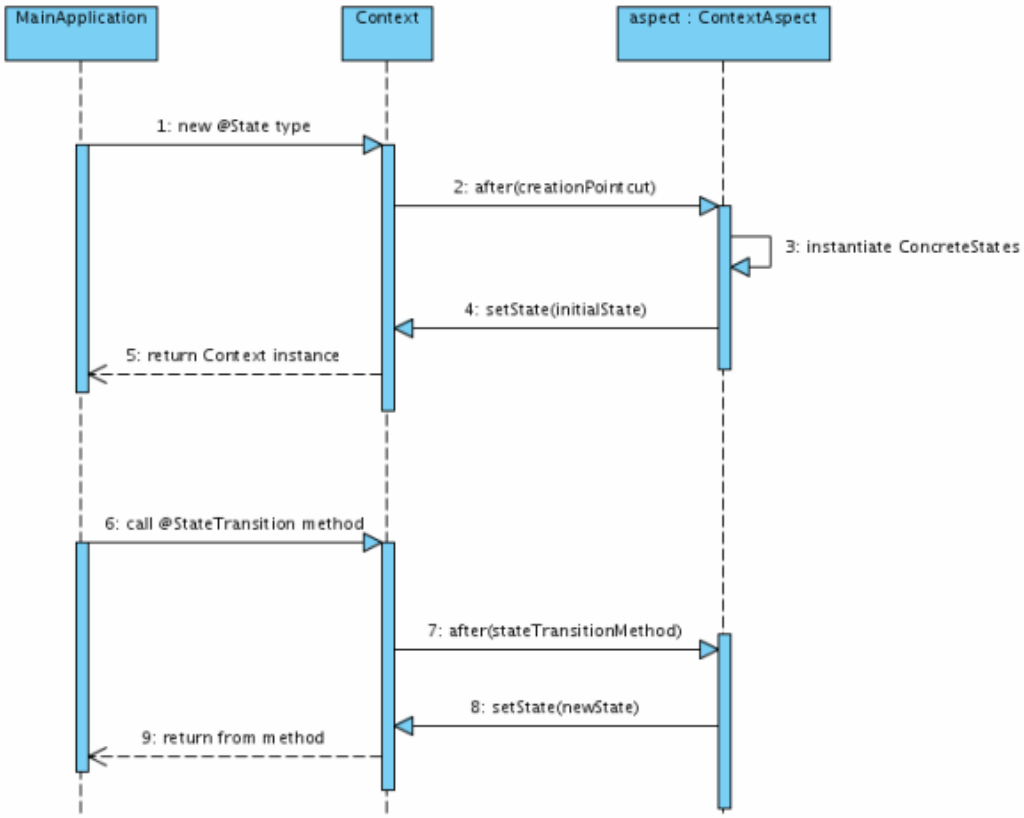

Figure 30: State Sequence Diagram

Annotations

State.java

```
package fi.joensuu.state.ann;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
/*
 * Annotation  for marking a class as playing the Context role
 * in the State pattern.
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface State {
  /*
   * Array of Class objects, representing the types of the ConcreteStates
   */
```

```
  java.lang.Class[] states();
  /*
   * Attribute representing the instantiation policy of the ConcreteStates
   */
  StateInstantiationPolicy instantiationPolicy() default
                                          StateInstantiationPolicy.EAGER;
}
```

StateInstantiationPolicy.java

```
package fi.joensuu.state.ann;

/*
 * Enum representing the available instantiation policies for
 * ConcreteStates.
 * EAGER = all instances are created when the Context class is instantiated
 * LAZY = instances of the ConcreteStates are created when are first needed
 */
public enum StateInstantiationPolicy {
  EAGER,
  LAZY
}
```

StateTransition.java

```
package fi.joensuu.state.ann;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

/*
 * Annotation marking a state transition triggering method.
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface StateTransition {
  /*
   * Attribute holding the type of the next ConcreteState
   */
  java.lang.Class nextState();
}
```

The annotations and the enumeration are used to configure state transitions and state instantiation policies for pattern's instances. *Class* objects are used to define states instead of String to take advantage of compile time checking of types.

Aspects

TCPStateAspect

```
package fi.joensuu.state.as;
import fi.joensuu.state.pattern.*;
import fi.joensuu.state.ann.*;
import java.lang.reflect.Method;
import java.util.*;
import org.aspectj.lang.reflect.MethodSignature;
```

```
/*
 * The State aspect. It intercepts the calls to the constructor
 * of the @State annotated classes, and creates instances for the
 * ConcreteStates. Also intercepts call to @StateTransition annotated
 * methods and sets the new current state.
 */

public aspect  TCPStateAspect {

  /*
   * Data structure managing the relationship between instances
   * of the Context class and its associated ConcreteStates.
   * In this example, the Context class is the TCPConnection,
   * and the State interface, TCPState.
   */
  private WeakHashMap<TCPConnection, Map<Class,TCPState> > dataStr =
      new WeakHashMap<TCPConnection, Map<Class,TCPState> >();
  /*
   * Pointcut for the interception of constructor
   * execution for @State annotated classes.
   */
  pointcut creation(TCPConnection cnx, State states) :
             execution (TCPConnection.new(..)) && this(cnx) && @this(states);


  /*
   * Pointcut for the interception of @StateTransition
   * annotated method calls.
   */
  pointcut stateTransitionMethod(TCPConnection cnx) :
        execution(@StateTransition * (@State *).*(..)) && this(cnx) ;


    /*
   * After advice, sets the new current state
   */
  after(TCPConnection cnx): stateTransitionMethod(cnx) {

    MethodSignature sig = (MethodSignature)thisJoinPointStaticPart.getSignature();
    Method met = sig.getMethod();
    Class next = (((StateTransition)(met.getAnnotations()[0])).nextState());
    cnx.setState(getStates(cnx).get(next));

  }

  /*
   * After advice, creates the instances of the ConcreteStates
   * and inserts them in the managing data structure
   */
  after(TCPConnection cnx, State states) : creation(cnx, states) {
    if(cnx != null) {
      Map<Class,TCPState> tmp = new WeakHashMap<Class,TCPState>();
      for(Class c : states.states()){
        try{
          tmp.put(c,(TCPState) c.newInstance());
        }catch(Exception ex){
          ex.printStackTrace();
        }

      }
      addData(cnx,tmp);
      cnx.setState(getStates(cnx).get(states.states()[0]));
    }else{
      System.out.println("null");
    }
  }


  /*
   * Method that adds a map of ConcreteStates for a Context instance
```

```
   */
  public void addData(TCPConnection cnx, Map<Class,TCPState> tmp){
    dataStr.put(cnx,tmp);
  }

  /*
   * Getting the map of ConcreteStates for a Context instance
   */
  public Map<Class,TCPState> getStates(TCPConnection cnx){
    return dataStr.get(cnx);
  }
}
```

The state aspect contains the pattern related code. In this case, the aspect is particular to the TCP connection and states example.

Pattern classes

TCPState.java

```
package fi.joensuu.state.pattern;
/*
 * The State interface, in this case TCPState
 */
public interface TCPState {
  void open();
  void send();
  void close();
}
```

TCPClosed.java

```
package fi.joensuu.state.pattern;
/*
 * Mock implementation of a ConcreteState
 */
public class TCPClosed implements TCPState {

  public void send() {
    // TODO Auto-generated method stub

  }

  public void close() {
    // TODO Auto-generated method stub
    System.out.println("CLosed: close");
  }

  public void open() {
    // TODO Auto-generated method stub
    System.out.println("CLosed: open");
  }

}
```

TCPEstablished.java

```
package fi.joensuu.state.pattern;
/*
 * Mock implementation of a ConcreteState
 */
public class TCPEstablished implements TCPState {

  public void send() {
    // TODO Auto-generated method stub
    System.out.println("Established: send");
```

```
  }

  public void close() {
    // TODO Auto-generated method stub
    System.out.println("Established: close");
  }

  public void open() {
    // TODO Auto-generated method stub
    System.out.println("Established: open");

  }

}
```

TCPState.java

```
package fi.joensuu.state.pattern;
/*
 * Mock implementation of a ConcreteState
 */
public class TCPListen implements TCPState {

  public void send() {
    // TODO Auto-generated method stub
    System.out.println("Listen: send");

  }

  public void close() {
    // TODO Auto-generated method stub
    System.out.println("Listen: close");

  }

  public void open() {
    // TODO Auto-generated method stub
    System.out.println("Listen: open");

  }

}
```

TCPConnection.java

```
package fi.joensuu.state.pattern;
import fi.joensuu.state.ann.*;
/*
 * The Context class, in this case TCPConnection
 */

@State(states={TCPClosed.class, TCPEstablished.class, TCPListen.class})
public class TCPConnection {
  private TCPState state;

  public TCPConnection(){

  }

  /*
   * state transition triggering method
   */
  @StateTransition(nextState=TCPListen.class)
  public void open(){

    state.open();
```

```
  }

  /*
   * state transition triggering method
   */
  @StateTransition(nextState=TCPClosed.class)
  public void close(){
    state.close();
  }

  /*
   * state transition triggering method
   */
  @StateTransition(nextState=TCPEstablished.class)
  public void send(){
    state.send();
  }

  public void setState(TCPState newState){
    System.out.println(newState);
    this.state = newState;

  }
}
```

StateMain.java

```
package fi.joensuu.state.pattern;
public class StateMain {
  /*
   * Main entry point, demo application
   */
  public static void main(String[] args) {
    // TODO Auto-generated method stub
    TCPConnection cnx =  new TCPConnection();
    cnx.open();
    cnx.send();
    cnx.close();
  }

}
```

This classes and interfaces represent the static part of the pattern. They play the role of the *Subject*, *State* and *ConcreteState*.

Conclusions

The State pattern, metadata and AOP are a good fit. An aspect is used for encapsulating the logic of the state transitions. The rules for the transitions are expressed as annotations. A class level annotation defines the types of the states supported by that *Context*. Each method that triggers a state change is marked with an annotation configuring the type of the next state. When Java will support variable level annotations, particular instances of the State pattern can be configured independently. The AOP and metadata implementation separates the logic of state transitions (aspects), the configuration of states and state transitions (annotations) and the logic performed by the pattern implementation (*Context*, *State*, *ConcreteState1*, *ConcreteState2*).

4.4 Proxy

Description

The Proxy pattern (see Figure 31) shows how an object can be hidden behind a placeholder or surrogate that exhibits the same interface as the original object. The proxy is an object that holds a reference to the real object and is used instead of it.

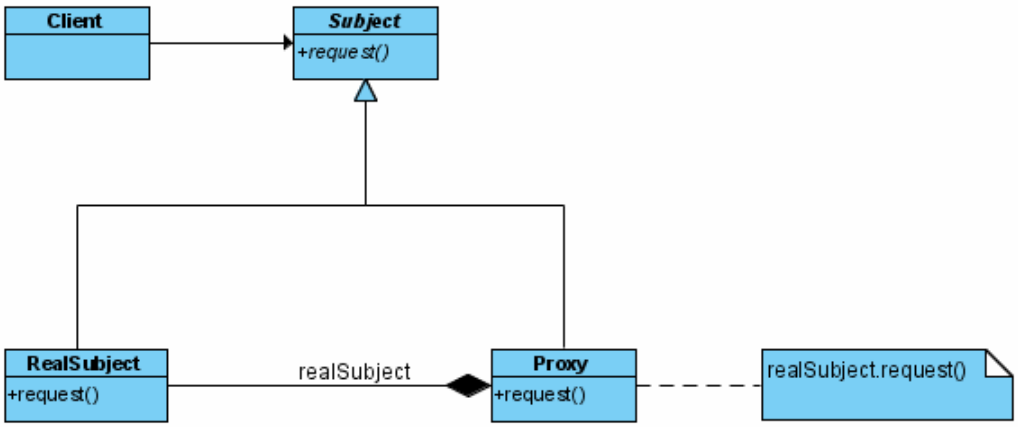

Figure 31: Proxy UML

Implementation details

The proxy pattern is used to accomplish different goals, though it has more or less the same structure. As the Iterator, this pattern becomes ubiquitous in almost all modern development platforms, in the form of a dynamic proxy [DynamicProxy].Proxy pattern implementations are heavily used in the development of run time weaving AOP frameworks. All the objects to be advised are hidden behind proxies, in which the advices' code resides. Dynamic proxies are general solutions for creating proxies for any class type. This flexibility comes with the price of complexity, decreased speed and verbosity; hence developers need sometimes to write their own proxy pattern implementations.

The goals the Proxy pattern tries to achieve are:

- Lazy loading of the original object.
- Method interception of the original object's methods in order to add behavior.
- The original object is a remote object, which the proxy makes it appear local.

Aspect Oriented Implementation

Method Interception proxies

Method interception proxies are a direct equivalent of an aspect with before, after and around advices on all the methods of a class. Depending on the need, different aspect instantiation policies can be used, like normal aspect, perthis or pertarget. The perthis and pertarget require a pointcut parameter to create an instance of the aspect for every joinpoint captured by the pointcut. The aspect will play the role of the proxy in a transparent manner. The method interception proxy is used in the implementation of the other two types of proxies.

Lazy loading of the original object

Lazy loading is delaying the creation of an object to as late as possible. This usually applies on objects that are expensive to create or require a lot of resources. Both the lazy object and the proxy implement the same interface. The straightforward AOP approach is an aspect creating an instance of either of them, as configured by an annotation. In case of proxy creation, the aspect intercepts the first method call requiring the real object, creates an instance of it, and injects it into the proxy. All subsequent method calls are forwarded to this instance. This approach requires intercepting the constructor call of the lazy object and returning a proxy instance instead. AspectJ forbids this scenario unless proxy is a subclass of the lazy object. This is not an option because a subclass of the lazy object will involve the same expensiveness as the superclass. One solution is to have a lightweight, cheap class implementing the same interface as the expensive, heavyweight class. The lightweight class is used instead of the heavyweight class in order to apply the method interceptor proxy aspect. The aspect needs an instance to be attached to. Annotations are used for configuring whether the lightweight object or the proxy is created. Requiring a special design is a drawback of using AspectJ for implementing a lazy loading proxy.

Remote proxy

The remote proxy hides the location of the real object, making transparent whether it is a distributed object or a local one. The benefits include easiness of testing by using mock local objects; and location transparency. The AOP implementation of the remote proxy is based on [RMIHello]. There are two parts in the pattern implementation: the client side and the server side.

Java RMI [RMI] involves specific code crosscutting the concerns of the remote object. The AOP implementation encapsulates this code.

On the server side, an aspect is used to capture the constructor call of the object to be exported as remotely available. The aspect contains the RMI specific code for exporting the object. An annotation is used to configure remote objects.

On the client side, an approach similar to the lazy loading proxy is used. The remote object is available as a local object. It is configured as remote using an annotation. The method intercepting proxy is attached to the local object. This proxy initializes a stub to the remote object using RMI specific code and forwards local method call to the remote object.

Source Code & Sequence Diagram

Method Interceptor Proxy

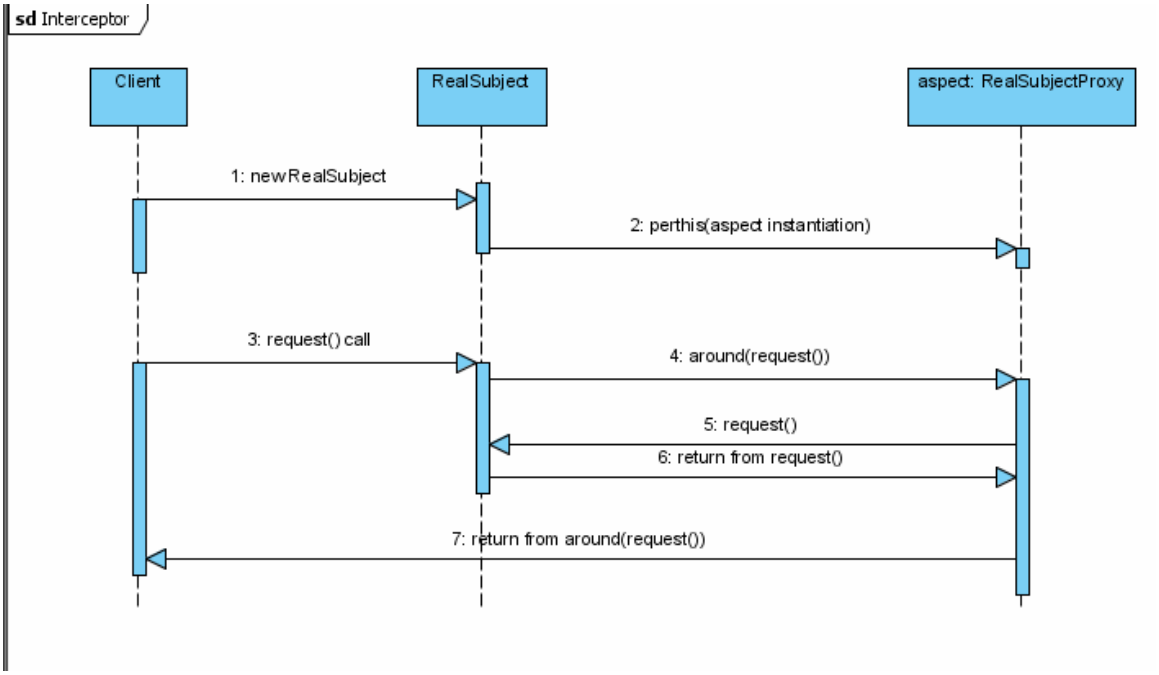

Figure 32: Method Interceptor Proxy Sequence Diagram

Subject.java

```
package fi.joensuu.proxy.interceptor.pattern;

/*
 * The Subject interface from the
 * Proxy pattern.
 */
public interface Subject {
```

```
  void request();
}
```

RealSubject.java

```
package fi.joensuu.proxy.interceptor.pattern;

import fi.joensuu.proxy.interceptor.ann.ProxyInterceptor;
import fi.joensuu.proxy.lazy.ann.LazyProxy;
import fi.joensuu.proxy.lazy.pattern.LazyRealSubject;


/*
 * RealSubject is the implementation
 * of the Subject interface.
 */
@ProxyInterceptor
public class RealSubject implements Subject {

  public void request() {
    // TODO Auto-generated method stub
    System.out.println(this + " request()");
  }

}
```

ProxyInterceptor.java

```
package fi.joensuu.proxy.interceptor.ann;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

/*
 * Annotation used to mark
 * method interceping proxies.
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface ProxyInterceptor {

}
```

ProxyInterceptorAspect.aj

```
package fi.joensuu.proxy.as;

/*
 * Abstract aspect. It uses a perthis aspect instantiation
 * policy associated with the subjectConstruction abstract pointcut.
 * Usually, subaspects define the subjectConstruction poincut
 * as the execution of the constructor of the class to be proxied.
 */
public abstract aspect ProxyInterceptorAspect perthis(subjectConstruction()){

       abstract pointcut subjectConstruction();
}
```

RealSubjectProxy.aj

```
package fi.joensuu.proxy.as;

import fi.joensuu.proxy.interceptor.ann.*;
import fi.joensuu.proxy.interceptor.pattern.*;
```

```
/*
 * A method intercepting proxy for RealSubject.
 */
public aspect RealSubjectProxy extends ProxyInterceptorAspect{

  /*
   * Instantiation of RealSubjects.
   */
  pointcut subjectConstruction() :
            execution ( (@ProxyInterceptor RealSubject).new(..));
  /*
   * Request method call intercepting pointcut.
   */
  pointcut requestCall() :
            execution (public void RealSubject.request());

  /*
   * Around advice, wrapping request method.
   */
Object around() : requestCall(){

    Object result = null;
    System.out.println(this +" before request()");
    result = proceed();
    System.out.println(this +" after request()");
    return result;
  }
}
```

The generic part of the pattern implementation consists of an abstract aspect (*ProxyInterceptorAspect*) and an annotation (*ProxyInterceptor*). The annotation is used for marking classes to be proxied. Each class needs a specific proxy aspect written for it. The benefit of the annotation is that by removing it from the class declaration, the class is not proxied. The annotation is the bind between the class and the proxy aspect. *ProxyInterceptorAspect* defines an instantiation policy (*perthis*) having as parameter an abstract pointcut (*subjectConstruction*). An aspect instance will be created for each joinpoint satisfying the pointcut. This is the proxy creation part of the pattern. *RealSubjectProxy* is a proxy for the *RealSubject* class. It extends the *ProxyInterceptorAspect*, defining the *subjectConstruction* pointcut as the constructor calls of the *RealSubject* class annotated with *ProxyInterceptor*. It also provides an around advice on *RealSubject*'s method request. This is the basic usage scenario: define the abstract pointcut as the execution of the real object's constructor and provide before, after or around advices on its methods.

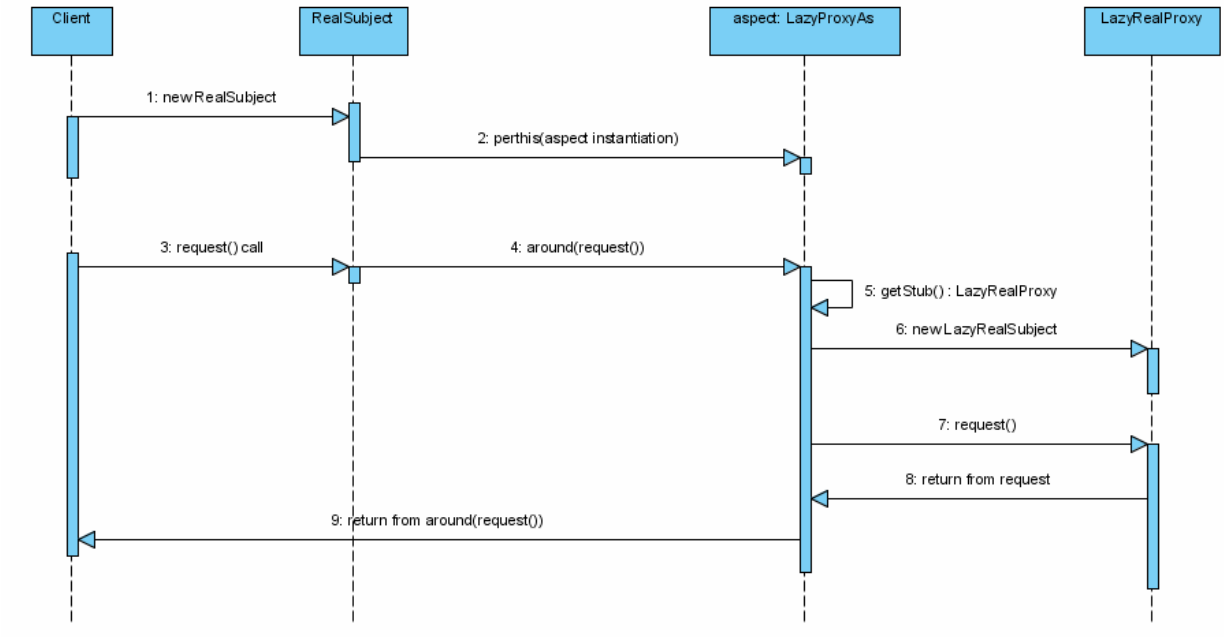

Figure 33: Lazy Initialization Proxy Sequence Diagram

LazyProxy.java

```
package fi.joensuu.proxy.lazy.ann;

import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

/*
 * Annotation for marking a class
 * as being a part of the lazy proxy
 * pattern.
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface LazyProxy {
  Class subjectType();
}
```

LazyRealSubject.java

```
package fi.joensuu.proxy.lazy.pattern;
import fi.joensuu.proxy.interceptor.pattern.Subject;


public class LazyRealSubject implements Subject{

  public LazyRealSubject(){

  }


  public void request() {
    System.out.println(this + " request()");
  }
}
```

RealSubject.java

```
package fi.joensuu.proxy.interceptor.pattern;

import fi.joensuu.proxy.interceptor.ann.ProxyInterceptor;
import fi.joensuu.proxy.lazy.ann.LazyProxy;
import fi.joensuu.proxy.lazy.pattern.LazyRealSubject;


/*
 * RealSubject is the implementation
 * of the Subject interface.
 */
@LazyProxy(subjectType = LazyRealSubject.class)
public class RealSubject implements Subject {

  public void request() {
    // TODO Auto-generated method stub
    System.out.println(this + " request()");
  }

}
```

LazyProxyAs.aj

```
package fi.joensuu.proxy.as;

import fi.joensuu.proxy.lazy.ann.*;
import fi.joensuu.proxy.interceptor.pattern.*;
import fi.joensuu.proxy.lazy.pattern.*;
import fi.joensuu.proxy.remote.client.ann.RMIProxy;

/*
 * Lazy initialization aspect. It extends the method intercepting aspect.
 */
public aspect LazyProxyAs extends ProxyInterceptorAspect{

  /*
   * Reference to the real object, which is lazy instantiated.
   */
  Subject stub = null;

  /*
   * Instantiation policy pointcut.
   *
   */
  pointcut subjectConstruction() :
             execution ( (@LazyProxy RealSubject).new(..));
  /*
   * Helper method, it checks if the reference to the lazy objects
   * exists, and if not, creates an instance of the lazy object.
   */
  private Subject getStub(Class subjType){
    if(stub == null){
      try {
        stub = (Subject)subjType.newInstance();
      } catch (InstantiationException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
      } catch (IllegalAccessException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
      }
    }
    return stub;
  }
```

```
  /*
   * Pointcut capturing request() method call.
   */
   pointcut requestCall() :
             execution (void RealSubject.request());

  /*
   * Around advice, calling the request method on the lazy object.
   */
   Object around() : requestCall(){

    Object res = null;
    Class subjType = thisJoinPoint.getThis().getClass()
            .getAnnotation(LazyProxy.class).subjectType();

    getStub(subjType).request();

    return res;
  }

}
```

ProxyMain.java

```
package fi.joensuu.proxy.Main;

import fi.joensuu.proxy.interceptor.pattern.RealSubject;
import fi.joensuu.proxy.interceptor.pattern.Subject;

public class ProxyMain {

  /**
   * @param args
   */
  public static void main(String[] args) {
    // TODO Auto-generated method stub
    Subject subj = new RealSubject();
    Subject subj2 = new RealSubject();
    subj.request();
    System.out.println("------------------");
    subj2.request();
    System.out.println("//////////////////////////////////");
    subj.request();
    System.out.println("------------------");
    subj2.request();

  }

}
```

The lazy loading proxy is a method interceptor proxy. The lightweight object is *RealSubject* while the heavyweight is *LazyRealSubject*. *RealSubject* is configured to be replaced by an instance of *LazyRealSubject* using the *LazyProxy* annotation. *LazyProxyAs* is the lazy loading proxy; it contains a reference to the heavyweight object. When the first method call is made to a *RealSubject* instance, an instance of the configured heavy object is created and subsequent method calls are forwarded to that instance.

Remote Proxy

Client

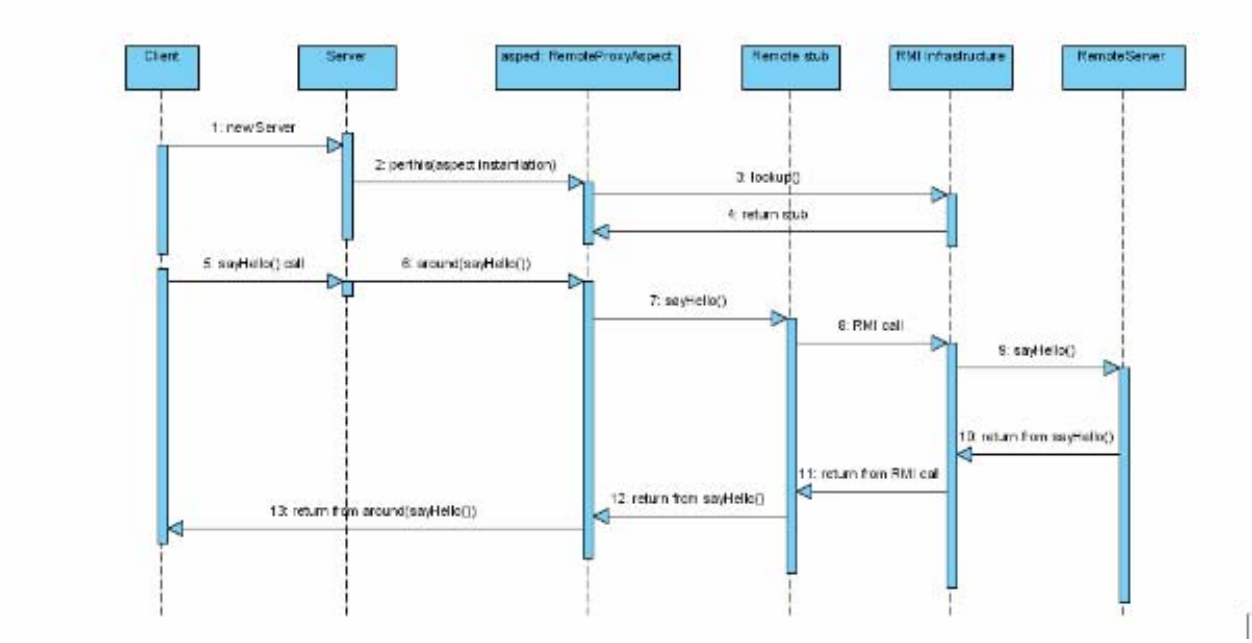

Figure 34: Remote Proxy Client Sequence Diagram

Hello.java

```
package fi.joensuu.proxy.remote.client.example.hello;

import java.rmi.Remote;
import java.rmi.RemoteException;


/*
 * The remote interface required by RMI
 */
public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

Server.java

```
package fi.joensuu.proxy.remote.client.example.hello;

import fi.joensuu.proxy.remote.client.ann.RMIProxy;


/*
 * The implementation of the Server.
 * If the @RMIProxy annotation is present,
 * it will accessed remotely, otherwise locally.
 */
@RMIProxy(host="localhost",name="Hello")
public class Server implements Hello {

  public Server() {
  }
```

```
  public String sayHello() {
    return "Hello, world!";
  }


}
```

RMIProxy.java

```
package fi.joensuu.proxy.remote.client.ann;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

/*
 * Annotation for marking a remote object
 * accessible using RMI
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface RMIProxy {
  /*
   * The host name of the server where the remote object
   * exists.
   */
  String host();
  /*
   * The name of the remote object.
   */
  String name();
}
```

RemoteProxyAspect.aj

```
package fi.joensuu.proxy.as;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

import fi.joensuu.proxy.remote.client.ann.*;
import fi.joensuu.proxy.remote.client.example.hello.*;


/*
 * RMIClient aspect. It extends the interceptor aspect.
 */
public aspect RemoteProxyAspect extends ProxyInterceptorAspect{

  /*
   * reference to the RMI stub
   */
  Hello stub = null;

  /*
   * Instantiation of the object for which a
   * RMI stub has to be created.
   */
 pointcut subjectConstruction() :
             execution ( public (@RMIProxy Server).new(..));
  /*
   * Around advice, creating the RMI stub.
   */
 after() : subjectConstruction(){
```

```
    String host = "";
    host = thisJoinPoint.getThis().getClass()
              .getAnnotation(RMIProxy.class).host();
    String name = "";
    name = thisJoinPoint.getThis().getClass()
              .getAnnotation(RMIProxy.class).name();
    try {
        Registry registry = LocateRegistry.getRegistry(host);
        Hello tmp = (Hello) registry.lookup(name);
        stub = tmp;

    } catch (Exception e) {
        System.err.println("Client exception: " + e.toString());
        e.printStackTrace();
    }
  }

  /*
   * Pointcut capturing sayHello method call.
   */
  pointcut sayHelloCall() :
       execution (String Server.sayHello(..));
  /*
   * Around advice, using the RMI stub to make a
   * RMI call on the remote object and returning the result.
   */
  Object around() : sayHelloCall(){
       Object res = null;
       try {
              res = stub.sayHello();
       } catch (RemoteException e) {

              e.printStackTrace();
       }

       return res;
  }

}
```

Client.java

```
package fi.joensuu.proxy.remote.client.example.hello;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {

    private Client() {}

    public static void main(String[] args) {

      Server server = new Server();
      String response = server.sayHello();
      System.out.println("response: " + response);


    }
}
```

The remote proxy on the client side is an implementation of the method intercepting proxy. A local object is needed for the method intercepting proxy aspect to attach to. It is required that the local object implements the same interface as the remote object. The *subjectConstruction*

pointcut intercepts the execution of the *RMIProxy* annotated *Server* class' constructor. The annotation also configures the remote object's name and location. The proxy contains an RMI stub to which it forwards method calls. The stub is created in an after advice, executed after the *subjectConstruction* pointcut.

Server

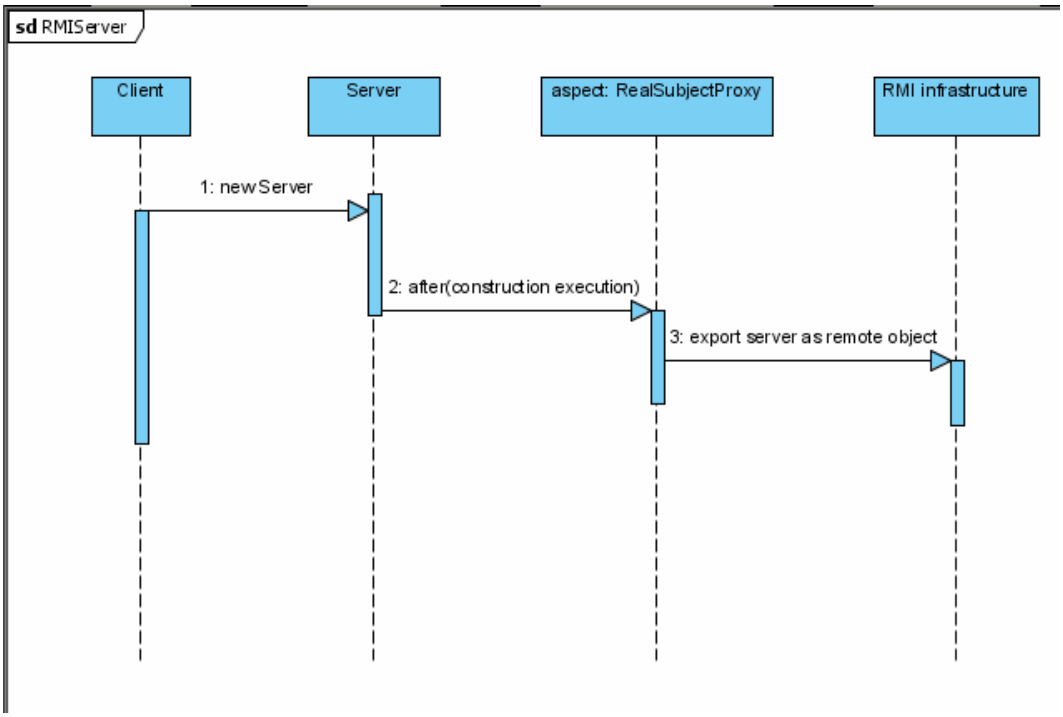

Figure 35: Remote Proxy Server Sequence Diagram

Server.java

```
package fi.joensuu.proxy.remote.server.example.hello;

import fi.joensuu.proxy.remote.client.example.hello.Hello;
import fi.joensuu.proxy.remote.server.ann.RMIServerExport;

/*
 * The Server object, this time exported from the server side
 * like a remote available object. The implementation of the method
 * is a little bit different, to show the difference between a local call
 * and a remote one.
 */
@RMIServerExport(name="Hello")
public class Server implements Hello {

  public Server() {
  }

  public String sayHello() {
    return "Hello, world! from the server";
  }

  public static void main(String args[]) {

    Server obj = new Server();
```

```
  }
}
```

RMIServerExport.java

```
package fi.joensuu.proxy.remote.server.ann;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

/*
 * Annotation used for exporting an object
 * as a remote object.
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface RMIServerExport {
  /*
   * Name of the remote object
   */
  String name();
}
```

RMIServerAspect.aj

```
package fi.joensuu.proxy.as;

import java.rmi.Remote;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

import fi.joensuu.proxy.remote.server.ann.RMIServerExport;
import fi.joensuu.proxy.remote.client.example.hello.*;



/*
 * Aspect exporting objects marked with @RMIServerExport annotation as RMI
 * remote objects.
 */
public aspect RMIServerAspect {

  /*
   * Pointcut capturing the creation of the annotated object.
   */
 pointcut serverCreation(Remote server) :
             execution (public (@RMIServerExport Remote+).new(..))
             && this(server);
  /*
   * After advice exporting the created object as
   * a remote object.
   */
  after(Remote server) : serverCreation(server){

    try {
      String name = Server.getClass().getAnnotation(RMIServerExport.class)
                          .name();
      Hello stub = (Hello) UnicastRemoteObject.exportObject(server, 0);

      Registry registry = LocateRegistry.getRegistry();
      registry.rebind(name, stub);

      System.err.println("Server ready");
```

```
    } catch (Exception e) {
      System.err.println("Server exception: " + e.toString());
      e.printStackTrace();
    }
  }
}
```

The server side involves an aspect (*RMIServerAspect*) that captures the constructor call of *RMIServerExport* annotated classes. It contains an after advice, woven after the constructor call, which exports those instances as RMI remote objects. *Server* is the POJO exported as a RMI remote object. It implements the *Hello* interface, an RMI remote interface. This relation can be managed using static crosscutting, hence making remote objects pure POJO's.

Conclusions

The proxy pattern is one way in which runtime weaving AOP frameworks are constructed. This pattern has several goals: lazy loading, method interception and location transparency (remote object appears local). A method intercepting proxy is easily implemented using an aspect. Annotations are used to mark the class to be proxied resulting in a declarative way of plugging/unplugging the pattern. When Java will support variable level annotations, it will be possible to proxy particular instances. Lazy loading of an object requires some special design of the class to be proxied. This happens because the lightweight object has to be created by default. The same is valid also for the remote object. AspectJ cannot wrap around the construction of an object and return a type that is not a subtype of the wrapped object. AOP and metadata separate the configuration of proxies (annotations), the logic to be performed in the proxy (aspects) and the original object.

5. Conclusions

AOP is a programming paradigm which comes as an extension to OOP to allow the encapsulation of crosscutting concerns. As OOP brought the concepts of class, method and attribute, AOP comes with its own set of concepts: pointcut, advice, introduction, aspect. Due to the fact that AOP comes not as a programming language, but as frameworks, in order to apply aspects a compiler-like entity is needed. This entity bears the name aspect weaver and the process is called weaving. The weaver is just one component of an AOP framework. The other one is the specific language used to express AOP specific constructs. Hence, in order to classify AOP frameworks, those two components have to be analyzed. Depending on when the weaving occurs, there are compile time, load time and run time frameworks. As for the specific language, there is a plethora of solutions, raging from XML files to language extensions. AspectJ is the most successful AOP framework to date. It offers the possibility of compile time or load time weaving. Its specific language is an extension to the Java programming language.

Design patterns are generic solutions to recurrent problems in object oriented design. The "Gof" patterns have the status of classics due to their generality and ubiquitousness. One of the most important achievements of these patterns is the creation of a common vocabulary between software engineers. These facts concur to make the "Gof" patterns a choice for proving new technologies. AOP aims to extend OOP making this choice even more evident. AspectJ was chosen to provide the aspect oriented implementation of the 23 "Gof" patterns.

The goal of this thesis is to use design patterns, AspectJ and metadata, in form of Java annotations, as proof for a solution to overcome two of the most important critics of AOP, namely the "tyranny of the dominant signature" and flow hiding. The tyranny of the dominant signature is the tight coupling of method or type signature to the weaving of aspects. Flow hiding is the lack of information for the developer on where and how aspects are woven. Annotations are used to mark joinpoints to be advised by aspects incorporating the pattern's logic. The results yield the following conclusion: in order to have a beneficial AOP implementation, pattern related code should crosscut the code performing the logic of the participants in the pattern. A significant number of the "Gof" pattern are either generic solutions (Facade, Interpreter) or pure object oriented solutions. The following four patterns offer the most beneficial implementations using AspectJ and annotations: Singleton, Observer, State and Proxy. There is a recurring theme in the

design of these patterns: annotations are used to mark and configure the participants, while the aspects hold the patterns' logic. By using annotations, the types involved in the pattern are loose coupled with the aspects. Also, plugging/unplugging the pattern resumes to marking/not marking types with annotations. All pattern related code is separated from the participants and has a higher degree of generality. Another important achievement is the lack of coupling to a specific AOP framework.

Further research directions

An important direction to be followed is the composition of patterns using metadata and AOP. Such an analysis is important in the context in which an object participates in multiple patterns. Work has been performed in this area, but without the use of metadata. Another significant direction is the analysis of metadata and AOP applied to patterns once local variable annotation would be available on the Java platform. The last, but not the least, important direction is the application of AOP and metadata in the implementation of domain specific patterns (e.g. remoting patterns, enterprise patterns).

References

All URL addresses were valid at May 11th 2008.

[Alexander77] Alexander C., Ishikawa S., Silverstein M., Jacobson M., Fiksdahl-King I., Angel S.: *A Pattern Language*. Oxford University Press, 1977.

[Annotations] Sun Microsystems: *Java Annotations*,
(http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html).

[AOSD] Aspect Oriented Software Development: *Aspect Oriented Software Development official website*, (http://aosd.net/).

[Aspect#] Castle Project: *Aspect#*, (http://www.castleproject.org/aspectsharp/index.html).

[AspectC] The Software Practices Lab: *AspectC*,
(http://www.cs.ubc.ca/labs/spl/projects/aspectc.html).

[AspectJ] The Eclipse Foundation: *AspectJ*, (http://www.eclipse.org/aspectj).

[Beck87] Beck K., Cunningham W.: Using Pattern Languages for Object-Oriented Programs, *OOPSLA '87 workshop on Specification and Design for Object-Oriented Programming*, 1987.

[Dijkstra82] Dijkstra E. W.: *On the role of scientific thought, Selected writings on Computing: A Personal Perspective*, Springer-Verlag, 1982.

[DynamicProxy] Sun Microsystems, *Java Dynamic Proxy*,
(http://java.sun.com/j2se/1.5.0/docs/api/java/lang/reflect/Proxy.html).

[Elrad01] Elrad T., Filman R. E., Bader A.: Aspect-oriented programming: Introduction, *Communications ACM* 44, 10 (Oct. 2001), pages 29-32, 2001.

[Fowler00] Fowler M., Parsons R., MacKenzie J.: *Plain Old Java Object*,
(http://www.martinfowler.com/bliki/POJO.html).

[Fowler04] Fowler M.: *Inversion of Control Containers and the Dependency Injection pattern*,
(http://martinfowler.com/articles/injection.html).

[Fox01] Fox J.: *When Singletons are not singletons*,
(http://www.javaworld.com/javaworld/jw-01-2001/jw-0112-singleton.html).

[Gamma95] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Hannemann02] Hannemann J., Kiczales G.: Design Pattern Implementation in Java and AspectJ, *Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 161-173, 2002.

[Gartner95] Gartner Inc.: *Understanding hype cycles*,
(http://www.gartner.com/pages/story.php.id.8795.s.8.jsp).

[JSE1.5] Sun Microsystems: *Java Programming Language 1.5*,
(http://java.sun.com/j2se/1.5.0/docs/guide/language/).

[JBossAOP] JBoss.org: *JBoss AOP official website*, (http://labs.jboss.com/jbossaop/).

[JEE] Sun Microsystems: *Java Enterprise Edition official website*, (http://java.sun.com/javaee/).

[Johnson04] Johnson R., Hoeller J.: *Expert One-on-One J2EE Development without EJB*, Wiley, 2004.

[JSR 308] Java Community Process: *Annotations on Java Types*,
(http://www.jcp.org/en/jsr/detail?id=308).

[Kiczales97] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C. V., Loingtier J., Irwin J.: Aspect-Oriented Programming, *ECOOP 1997*, pages 220-242, 1997.

[Laddad03] Laddad R.: *AspectJ in Action*, Manning Publications, 2003.

[Laddad05] Laddad R.: *AOP and metadata: A perfect match, pt 2*,
(http://www.ibm.com/developerworks/java/library/j-aopwork4/index.html).

[Miles04] Miles R.: *AspectJ Cookbook*, O'Reilly, 2004.

[Olsen07] Olsen R.: *Design Patterns in Ruby*, Addison-Wesley, 2007.

[Pawlak05] Pawlak R., Retaillé J., Seinturier L.: *Foundations of AOP for J2EE Development*, APress, 2005.

[Ruby] Ruby Programming Language: *Ruby Programming Language*,
(http://www.ruby-lang.org/en/).

[RMI] Sun Microsystems: *Java Remote Method Invocation*,
(http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp).

[RMIHello] Sun Microsystems, *Java Remote Method Invocation Hello World*,
(http://java.sun.com/j2se/1.5.0/docs/guide/rmi/hello/hello-world.html).

[Spring] The Spring Source, *The Spring Framework*, (http://www.springframework.org/).

[SpringAOP] Spring Source, *Spring AOP documentation*,
(http://www.springframework.org/docs/wiki/Spring_AOP_Framework.html).