BACHELOR PROJECT REPORT

Vrije Universiteit Amsterdam

Faculty of Sciences

Division of Mathematics and Computer Science

Department of Information Management and Software Engineering

Supervisor	**Candidate**
Dr.-Ing. Ralf Lämmel	Jorrit N. Herder
ralf@cs.vu.nl	jnherder@cs.vu.nl

Abstract

Design patterns are well-established in providing design expertise in programming, in particular in object-oriented programming (OOP). Aspect-oriented programming (AOP) is an emerging programming paradigm based on the principle of modularization of crosscutting concerns. Because AOP languages are generally based on and provide extensions to OOP languages, existing object-oriented patterns possibly can be implemented more modular with AOP technologies or may be replaced by aspect-oriented patterns that provide a different solution for the same problem. The effects of AOP on existing object-oriented design patterns were studied in this bachelor project. For this, the Java and AspectJ pattern implementations by Jan Hannemann and Gregor Kiczales [4] were used as a starting point. The major part of the project was spent on a comparison of five of these pattern implementations to investigate whether the implementation structure changes or not and what benefits are obtained with an AspectJ implementation. The following design patterns were studied: observer, composite, visitor, decorator and proxy.

1 Introduction

Design patterns [1] are proven solutions that provide elegant ways for solving frequently occurring problems. They capture experience in a way that it is possible for others to reuse this experience. Design patterns are well-established in object-oriented programming (OOP), but also exist for other programming paradigms.

Aspect-oriented programming (AOP) [2] is an emerging programming paradigm that provides language mechanisms to capture the crosscutting structure of certain properties, which are called aspects. The modularization of aspects allows for simpler code that is easier to develop and maintain, and that has greater potential for reuse.

Just as OOP builds on ideas previously used in other programming paradigms, like procedural programming, AOP does not reject existing technology but extends it with possibilities to capture crosscutting concerns. An example AOP language is AspectJ [15], which extends Java [14] with aspect-oriented constructs. Because AOP languages are generally based on and provide extensions to OOP languages, existing object-oriented patterns possibly can be implemented more efficiently with AOP technologies or may be replaced by aspect-oriented patterns that provide a different solution for the same problem.

During this bachelor project it was studied how existing object-oriented design patterns evolve in the presence of AOP. As both design patterns and AOP are popular topics, related work was done elsewhere at the same time. Work by Jan Hannemann and Gregor Kiczales [3] was published halfway the project. This work has been used as a starting point to investigate language dependence of design patterns. Several design pattern implementations [4] in Java and AspectJ are compared to study the effects of AOP on them.

The rest of this bachelor project report is organized as follows. The next two subsections provide some information on what design patterns precisely are and discuss the language constructs that are used in the AspectJ pattern implementations. In Section 2, the work by Jan Hannemann and Gregor Kiczales, which was further investigated in this project, and earlier work by Peter Forbrig and Ralf Lämmel, which already indicated how design patterns can be reused, are discussed. In Section 3, it is studied whether the classification of design patterns in the *pattern space* [1] correlates with the benefits that are obtained with aspect-oriented implementations or not. In Section 4, the Java and AspectJ implementation structure of several design patterns is treated in depth – differences in the implementation structure and points of improvement are pointed out. In section 5, an overview of the results is given and conclusions are drawn.

1.1 What are design patterns?

Design patterns are well-established in providing design expertise in programming. The primary focus of patterns is not on technology, but rather is on documenting general design principles. As design patterns have become very popular, there exist numerous descriptions of what design patterns actually are. To prevent inventing yet another description, the reader is referred to the Gang of Four design pattern book [1]. This work provides the following statements, on what design patterns are:

- "Design patterns name, abstract, and identify the key aspects of common design structures that make it useful for *creating reusable object-oriented design*."
- "Design patterns are descriptions of communicating objects and classes that are customized to *solve a general design problem* in a particular context."
- "Each design pattern systematically names, explains and evaluates an important and *recurring design in object-oriented systems*."

Design patterns thus are a way to share a general solution for design problems that are encountered over and over again during the development of software systems. Design patterns are often mentioned in relation to OOP, but they are valuable in other programming paradigms as well. An interesting issue is how existing object-oriented design patterns evolve in the presence of other programming paradigms. In this project report, the effects of AOP on several object-oriented design patterns are studied.

1.2. Aspect-oriented programming with AspectJ

In this subsection, the principles of AOP with AspectJ that are of interest for this project are discussed; the reader is referred to [10,2,8,9] for a more detailed treatment of AspectJ's language constructs. AspectJ is an aspect-oriented extension to the Java programming language. AspectJ captures crosscutting concerns by defining special classes that can crosscut several other Java classes. These special classes are called *aspects*, just as the crosscutting concerns that they capture. Aspect declarations are similar to class declarations in Java and

can include any declaration permitted inside class declarations plus the aspect-oriented language constructs of AspectJ.

For some of the GoF design patterns, a reusable abstract aspect could be created that serves as a protocol [4]. The abstract aspect usually contains the roles that are assigned to the pattern's participants, but other properties, such the mapping between participants and facilities to implement needed functionality, also may be included. In order to use the design pattern, a concrete instance of the abstract aspect should be created to make all general properties specific.

Basically, two language constructs of AspectJ that are often used in relation to design patterns: *introduction* to assign roles and attach methods to the participants and *join points* to trigger certain events. Below, a short discussion is given of these how these principles are used to implement design patterns.

1.2.1 Assigning roles to participants

In most of the cases, different roles that are assigned by design patterns are modeled with empty interfaces in the abstract aspect. The assignment of roles to participants is done with AspectJ's *open class mechanism*. In a concrete pattern instance, the parents of the participating classes are declared. This means that the role interfaces are introduced into existing classes on compilation. Suppose an interface `Role` is defined in the abstract aspect, then it could be introduced into, i.e. the role `Role` could be assigned to, an existing class `Participant` with the following statement in a concrete pattern instance:

```
declare parents: Participant implements Role;
```

1.2.2 Attaching pattern functionality

Optionally some methods can also be introduced into the participants of the design patterns. This way, the functionality needed for the design pattern can be attached to the participants. This is, for example, needed with the visitor design pattern, where elements of the data structure must be prepared to accept visitors. Suppose a method `accept(Visitor v)` is needed in participants that are assigned the role `Node` in order to accept `Visitor`s, then the following AspectJ code can be used in an abstract pattern to introduce it:

```
public void Node.accept(Visitor v) {
    // call-back the visitor
    v.visit(this)
}
```

1.2.3 Event handling

For certain design patterns, such as the observer pattern, it is needed to capture events in the program flow to trigger some action. This can be done with *pointcuts*. Pointcuts can identify one or more *join points*, i.e. points interest in the execution of a program. Each join point can be wrapped with extra code that is run just before or just after the join point is reached. The attachment of extra code to be executed at join points is done in so-called *advice declarations*.

With AspectJ it is possible to create abstract pointcuts that can be used with advice declarations in abstract patterns aspects. This way, a general pattern aspect can be created, which only requires a concrete definition of the abstract pointcut. For example, in the abstract observer aspect an advice declaration tells to notify all observers just after reaching an abstract pointcut. If the participants are assigned the roles `Subject` and `Observer`, then an

abstract pointcut `event(Subject s)` that triggers an *after* advice declaration to notify all registered `Observers` can be defined in the abstract aspect as follows:

```
abstract pointcut event(Subject s);

after(Subject s): event(s) {
    // code to notify all registered observers
    ...
}
```

All that has to be done in the concrete pattern instance is to provide a concrete definition of the events that must be observed. If, for example, the events of interest are calls to the method `update()` in a `Participant` that has been assigned the role `Subject`, then this can be done as follows:

```
pointcut event(Subject s):
    call(void Participant.update()) && target(s);
```

2 Discussion of related work

In this section, some related work is discussed. As mentioned in the introduction, the work that was published by Jan Hannemann and Gregor Kiczales [3] was used for the investigation how existing object-oriented design patterns evolve in the presence of AOP. In that work, all GoF patterns were implemented in both Java and AspectJ and it was found that AspectJ improves the implementation of many design patterns. For 12 out of the 23 GoF patterns, a substantial part of the implementation could be abstracted into reusable code, which resulted in a library with abstract pattern aspects.

Previous work by Peter Forbrig and Ralf Lämmel [5,6] already indicated, "how reuse of patterns can be modeled based on superposition of class structures, and how schemes of reuse can be captured by a new kind of abstraction". In that work, an experimental programming language, *PaL* (*Pa*ttern *L*anguage), was used to develop a library covering all 23 GoF patterns. Although the programming language constructs of *PaL* and AspectJ to superimpose structure on the participants of design patterns are slightly different, the basic principles used to create reusable pattern implementations are similar.

In Subsection 2.1, the work of Jan Hannemann and Gregor Kiczales is introduced in more detail. Especially how the kinds of roles assigned by design patterns correlate with the benefits obtained with AspectJ implementations. In Subsection 2.2, the work by Peter Forbrig and Ralf Lämmel to create a reusable pattern library is introduced. Subsection 2.3 briefly compares the AspectJ and *PaL* approaches.

2.1 Design pattern implementation in Java and AspectJ

A comparison of Java and AspectJ implementation of GoF design patterns by Jan Hannemann and Gregor Kiczales shows that the implementation of design patterns is language-dependent. Compared to Java, AspectJ improves the implementation of many patterns. "In some cases this is reflected in a new solution structure with fewer or different participants, in other cases, the structure remains the same, only the implementation changes [3]". Improvements obtained with AspectJ are manifested in terms of better modularity. This was measured by looking at the code locality, reusability, composability, and (un)pluggability.

A correlation between the kinds of roles a pattern assigns to its participants and how much the pattern implementation benefits from AspectJ was found. For this, a distinction between *defining* and *superimposed* roles was made. A participant's role is defining if the participant has no functionality outside the pattern. A role is superimposed if the participant to which it is assigned has functionality and responsibility outside the pattern. Participants with a defining role are often accessible from clients, whereas participants with a superimposed role usually are not. The distinction between defining and superimposed roles is not always totally clear.

The pattern implementations in Java and AspectJ showed that the patterns with only superimposed roles benefit most from AspectJ. The AspectJ implementation of these patterns resulted in code locality, reusability, composability, and (un)pluggability. As superimposed roles lead to *crosscutting* among patterns and participants in several ways and AspectJ is intended to modularize crosscutting structure, the results should not be surprising: AspectJ modularizes the crosscutting that originates in superimposed roles. Three different kinds of crosscutting among participants and patterns were distinguished, including roles crosscutting participant classes.

In Table 2.1, the different kinds of roles that patterns assign to their participants are shown, together with a rough indication of the modularity obtained with AspectJ; the exact correlation between roles and modularity of the AspectJ implementations is shown in Table 3.2 in Subsection 3.2.

	Modularity	**Kinds of Roles** [1]	
	with AspectJ	*Defining*	*Superimposed*
Façade	Same as Java	Façade	-
Abstract Factory	No	Factory, Product	-
Bridge		Abstraction, Implementer	-
Builder		Builder, (Director)	-
Factory Method		Product, Creator	-
Interpreter		Context, Expression	-
Template Method	Partly	(Abstract), (Concrete Class)	(Abstract), (Concrete Class)
Adapter		Target, Adapter	Adaptee
State		State	Context
Decorator		Component, Decorator	Concrete Component
Proxy		(Proxy)	(Proxy)
Visitor		Visitor	Element
Command		Command	Commanding, Receiver
Composite		(Component)	(Composite, Leaf)
Iterator		(Iterator)	Aggregate
Flyweight		Flyweight Factory	Flyweight
Memento		Memento	Originator
Stategy		Strategy	Context
Mediator	Yes	-	(Mediator), Colleague
Chain of Resp.		-	Handler
Prototype		-	Prototype
Singleton		-	Singleton
Observer		-	Subject, Observer

[1] If the distinction between the kinds of roles was not totally clear, roles names are shown in parentheses.

Table 2.1: The modularity that was obtained with pattern implementations in AspectJ and the different kinds of roles that patterns assign to their participants. From this table a correlation between the modularity obtained with AspectJ and the kinds of roles becomes clear. This table was adopted from [3].

2.2 Design pattern implementation in *PaL*

In [3], the language support of AspectJ to modularize crosscutting concerns was used to create a reusable catalogue with 12 out of all 23 GoF design patterns. The AspectJ language constructs that were used to implement design patterns in were already introduced in Subsection 1.2. The most important construct was *introduction* to assign roles and attach methods to the participants. In [5,6], the experimental pattern language *PaL* was used to implement a library covering all 23 GoF patterns. The *PaL* language constructs that were used to create the library and to use it in programming are *nested classes* and a kind of *superposition* for class structures. These two constructs are discussed in the next two paragraphs.

Nested classes are used to capture the solution structure of a design pattern. Nested classes can encapsulate both abstract and concrete participants of a design pattern in a single class. For example, the PVISITOR class that is shown in Figure 2.1 models the visitor design pattern. The four nested classes in the figure are `ELEMENT` and `VISITOR`, defining the roles assigned by the pattern, and `CONCRETE_ELEMENT` and `CONCRETE_VISITOR`, identifying concrete participants.

```
class PVISITOR

  class ELEMENT
    feature
      accept(a_visitor: VISITOR)
      is deferred
    end
  end -- ELEMENT

  class CONCRETE_ELEMENT
    feature
      accept(a_visitor: VISITOR) is do
        a_visitor.visit_
          concrete_element(current)
    end
  end -- CONCRETE_ELEMENT

  class VISITOR
    feature
      visit_concrete_element
        (an_element: CONCRETE_ELEMENT)
      is deferred
    end
  end -- VISITOR

  class CONCRETE_VISITOR
    inherit VISITOR end
  end -- CONCRETE _VISTOR

end -- PVISITOR
```

```
class AbstractCompositeVisitor

   reuse PCOMPOSITE;

   PVISITOR rename
      ELEMENT as COMPONENT
      CONCRETE_ELEMENT as LEAF
      VISITOR.visit_concrete_element
         as visit_leaf

   PVISITOR rename
      ELEMENT as COMPONENT
      CONCRETE_ELEMENT as COMPOSITE
      VISITOR.visit_concrete_element
         as visit_composite
```

Figure 2.1: The visitor design pattern in *PaL*. The class `PVISITOR` models the visitor design pattern by means of nested classes. The class `AbstractCompositeVisitor` reuses the class `PCOMPOSITE` (not shown here) and superimposes the structure of the `PVISITOR` class on the nested classes `LEAF` and `COMPOSITE` of `PCOMPOSITE`. The examples `PVISITOR` and `AbstractCompositeVisitor` were taken from [5,6].

The essential language construct of *PaL* that allowed using classes in the pattern library, is the *reuse-clause*. The structure of classes that are reused is superimposed on the class that contains the reuse-clause. This way, certain functionality can be introduced in classes. Roles can be assigned by means of *superposition*. Superposition is a way to merge classes. In Figure

2.1, the `AbstractCompositeVisitor` is defined by merging the nested classes from the visitor and composite design pattern: the `PVISITOR` pattern classes is used two times in order to merge its nested class `CONCRETE_ELEMENT` with both `LEAF`s and `COMPOSITE`s. In actual programming, the class structure of patterns can be reused in the same way.

2.3 Comparison of AspectJ and *PaL*

The works described in [3] and [5,6] are similar in that both describe how a library with abstract patterns can be created and how these abstract patterns can be used in actual programming. The pattern classes in *PaL*, such as `PVISITOR`, can be compared to the abstract pattern aspects in AspectJ, such as `VisitorProtocol` that is discussed in Subsection 4.3. Reusing the *PaL* pattern classes basically is the same as assigning roles with AspectJ. In both cases some structure is superimposed on the participants of the design pattern. Superposition seems to be an essential language construct for creating a reusable pattern catalogue.

Although *PaL* and AspectJ are similar, there is a number of differences also. For example, *PaL* does not support the concept of join points, meaning that it is not possible to identify execution points of interests and handle event as easy as in AspectJ. Event handling is important for some design patterns, such as the observer pattern that is discussed in Subsection 4.1. Another difference lies in the acceptance of both languages. *PaL* is an experimental language based on a compilation to Eiffel, whereas AspectJ is one of the most important AOP languages currently available and is supported by a large user community.

During this bachelor project, the pattern language *PaL* was not investigated any further. It was interesting however, to see that some of the concepts that were used in the *PaL* and AspectJ implementations are similar. This work further focusses on the design pattern implementations in Java and AspectJ. In other work, the concepts underlying *PaL*, such as superimposition, have been integrated into Rational Rose [7]. Meanwhile, similar approaches were integrated into several case systems.

3 Classification of design patterns according to the pattern space

In Subsection 2.1, it was mentioned that a correlation between the benefits of AspectJ implementations and the kinds of roles patterns assign to their participants was found. It was therefore proposed to divide design patterns into three groups, namely: (a) those with only defining roles, (b) those with only superimposed roles, and (c) those with both kinds of roles. Design patterns in which the participants only have superimposed roles benefit most from an AspectJ implementation, as can be seen in Table 3.2 at the end of this section.

In [1], another way to divide design patterns into groups was mentioned, namely the *pattern space*. In Subsection 3.1, the pattern space is shortly introduced. After this, it is shown in Subsection 3.2 that the classification of design patterns in the pattern space is not correlated to benefits that can be obtained with AspectJ.

3.1 The pattern space

In the pattern space, design patterns are classified according to their *purpose* and their *scope*. The former criterion, purpose, reflects what a pattern does and is either creational, structural, or behavioral:

- *Creational patterns*: abstract the process of object instantiation;

- *Structural patterns*: deal with composition of classes or objects into larger structures;
- *Behavioral patterns*: define how classes or objects interact and distribute responsibility.

The latter criterion, scope, specifies whether a design pattern applies to classes or to objects:

- *Class patterns*: deal with static relationships between classes and subclasses established through inheritance, i.e. fixed at compile-time.
- *Object patterns*: deal with object relationships that can be changed at run-time and are more dynamic.

With three different purposes and two different scopes, six different groups of design patterns are to be distinguished: class creational, object, creational, class structural, object structural, class behavioral and object behavioral. In Table 3.1, all GoF design patterns are shown in the design pattern space.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibililty Command Iterator Mediator Memento Observer State Strategy Visitor

Table 3.1: Classification of all GoF patterns in the design pattern space. This table was adopted from [1].

3.2 Correlation of AspectJ benefits and the pattern space

The notion of a pattern space gave rise to the question whether the classification of design patterns is somehow correlated to the benefits that are obtained with an AspectJ implementation – compared to a Java implementation. In order to answer this question, the classification of each design pattern was compared with the modularity improvements that were found in [3]. The results in Table 3.2 show that there is *no correlation* between the classification in the pattern space and the benefits obtained with AspectJ. This holds for both dimension, purpose and scope, of the pattern space.

The reason that no correlation is found is not directly obvious. For *scope* on the one side this is rather easy to see, as scope has to do with the question whether a pattern has to do with static or dynamic relationships and not with the kind of – potentially crosscutting – structure that the relationship exactly entails. For *purpose* on the other hand it is not that obvious, because the qualifiers creational, structural and behavioral on first sight seem to be related with the kind of structure patterns have. The purpose of a pattern gives an indication whether is has a crosscutting structure – for which AspectJ has modularization constructs – and one thus might expect a correlation. The superimposed behavior on the participants, however, turns out to be more important in terms of crosscutting, as is illustrated in the next paragraph.

For example, creational patterns have to do with the process of object instantiation. This is something that happens a single time at a single place for each object, and thus seems not to have a crosscutting structure. The results in Table 3.2, however, show that creational patterns exist that do benefit from AspectJ. The crosscutting structure that is modularized by AspectJ in these cases originates from the behavior that is superimposed on the participants.

	Modularity Properties [1]				**Roles**		**Pattern Space** [2]	
	Locality	*Reusability*	*Composition Transparency*	*(Un) Pluggability*	*Defining*	*Super-imposed*	*Purpose*	*Scope*
Façade	Same implementation for Java and AspectJ				+	-	S	O
Abstract Factory	no	no	no	no	+	-	C	O
Bridge	no	no	no	no	+	-	S	O
Builder	no	no	no	no	+	-	C	O
Factory Method	no	no	no	no	+	-	C	C
Interpreter	no	no	n/a	no	+	-	B	C
Template Method	(yes)	no	no	(yes)	+	+	B	C
Adapter	yes	no	yes	yes	+	+	S	C/O
State	(yes)	no	n/a	(yes)	+	+	B	O
Decorator	yes	no	yes	yes	+	+	S	O
Proxy	(yes)	no	(yes)	(yes)	+	+	S	O
Visitor	(yes)	yes	yes	(yes)	+	+	B	O
Command	(yes)	yes	yes	yes	+	+	B	O
Composite	yes	yes	yes	(yes)	+	+	B	O
Iterator	yes	yes	yes	yes	+	+	S	O
Flyweight	yes	yes	yes	yes	+	+	S	O
Memento	yes	yes	yes	yes	+	+	B	O
Stategy	yes	yes	yes	yes	+	+	B	O
Mediator	yes	yes	yes	yes	-	+	B	O
Chain of Resp.	yes	yes	yes	yes	-	+	B	O
Prototype	yes	yes	(yes)	yes	-	+	C	O
Singleton	yes	yes	n/a	yes	-	+	C	O
Observer	yes	yes	yes	yes	-	+	B	O

[1] The use of parentheses means that some restrictions apply to the modularity properties. Also see [3].

[2] The pattern purpose can be Structural, Behavioral, or Creational. The pattern scope can be Class or Object.

Table 3.2: Design patterns grouped according to the way their implementation is affected by AspectJ. From the table becomes clear that (1) there is a correlation between the roles a pattern assigns and how much its implementation benefits from AspectJ and that (2) there is no correlation between the pattern space and the modularity properties found with an AspectJ implementation. The modularity properties and kinds of roles assigned by the patterns were taken from [3].

4 Comparison of design pattern implementations in Java and AspectJ

As mentioned in Subsection 2.1, most of the AspectJ design pattern implementations were an improvement over Java implementations. It was found that the solution structure changed for some of the design patterns, whereas it remained the same for other. For certain patterns several solution structures were possible. Unfortunately, it was not discussed in detail for which of the design patterns the solution structure changed – and in what way. Therefore, the Java and AspectJ solution structures of five design patterns were compared in this project. The implementations discussed here, including the code fragments, have been taken from [4].

The design patterns of which the Java and AspectJ implementations are compared include observer, composite, visitor, proxy, and decorator. For each of these patterns, a short description taken from [1] is given and the recurring problem is highlighted, after which the solution structure of the Java and AspectJ implementations is discussed and compared. The comparisons can be found in Subsection 4.1 to Subsection 4.5.

4.1 Observer

4.1.1 Intent

GoF: "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically [1]".

4.1.2 Recurring problem

With object-oriented programming, but other programming paradigms as well, a problem is often split into several components that work together to solve it. As the components are cooperating to solve the problem, related components must remain in a consistent state. Tightly coupling components to achieve consistency is unwanted, because it makes maintenance hard and reduces the reusability of individual components.

4.1.3 Structure of the Java solution

The observer design pattern helps to decouple components while retaining consistency. The observer pattern assigns two roles, namely *subject* and *observer*. Observers that depend on a certain subject register themselves to receive notifications of important changes. If such a notification is received, it is the observer's responsibility to process it in order to remain in a consistent state. This way the subject does not have to know the structure of its observers.

To allow observers to register for notification of certain events, subjects must implement the interface `Subject` and implement methods `attach(Observer o)` and `detach(Observer o)`. Some local data structure must be used to keep track of all registered observers. If a certain event takes place in the `Subject`, it notifies all registered `Observers`. The interface `Subject` is shown in Figure 4.1.1.

```
33  public interface Subject {
..
41      public void attach(Observer o);
..
49      public void detach(Observer o);
..
55      public void notifyObservers();
56  }
```

Figure 4.1.1: The interface `Subject` that must be implemented by classes that need to be observed. The `attach` (line 41) and `detach` (line 49) prototypes are respectively needed to register and unregister `Observers` for notification of certain events; a local data structure should be maintained for this. The prototype `notifyObservers` (line 55) is called by the `Subject` in case the events of interest take place.

Each observer must implement the interface `Observer` that contains the method `update(Subject s)`. The subject calls this method after the event takes place, so that the observer knows something happened. The `Subject` passes itself as an argument so that the `Observer` can find out what exactly happened and can do the necessary things to remain in a consistent state. The interface `Observer` is shown in Figure 4.1.2.

```
32  public interface Observer {
..
..    /* @param s the subject triggering the update */
41    public void update(Subject s);
42  }
```

Figure 4.1.2: The interface `Observer` that must be implemented by all observers. Classes that implements the interface `Observer` can register itself with a `Subject` for notification of certain events. The responsibility for taking the appropriate actions if the `Subject` calls the method `update` (line 41) lies with the individual `Observer`s.

4.1.4 Structure of the AspectJ solution

Both the subject and observer roles assigned by the observer pattern are *superimposed*. As mentioned in Subsection 2.1, patterns with only superimposed roles showed most improvements. With AspectJ, the entire observer design pattern can be abstracted into reusable code. The AspectJ implementation uses an abstract aspect `ObserverProtocol` that includes the `Observer` and `Subject` roles as empty interfaces, the mapping between the participants, an abstract pointcut to identify events in which observers are interested, and the code to notify and update all observers afterwards. See Figure 4.1.3.

```
 61  public abstract aspect ObserverProtocol {
 ..
 ..      /* Empty interface to model the subject role. */
 69      protected interface Subject  { }
 ..
 ..      /* Empty interface to model the observer role. */
 77      protected interface Observer { }
 ..
 ..      /* Locally stores mapping between Subjects and Observer */
 86      private WeakHashMap perSubjectObservers;
 ..
118      public void addObserver(Subject s, Observer o) {
119          getObservers(s).add(o);
120      }
...
130      public void removeObserver(Subject s, Observer o) {
131          getObservers(s).remove(o);
132      }
...
...      /* Abstracts the join points after which to do the update. */
140      protected abstract pointcut subjectChange(Subject s);
...
...      /* Advice declaration to notify Observers after event */
148      after(Subject s): subjectChange(s) {
149          Iterator iter = getObservers(s).iterator();
150          while ( iter.hasNext() ) {
151              updateObserver(s, ((Observer)iter.next()));
152          }
153      }
...
...      /* Method that abstracts how to update each Observer. */
163      protected abstract void updateObserver(Subject s, Observer o);
164  }
```

Figure 4.1.3: The abstract aspect `ObserverProtocol` includes the `Observer` and `Subject` roles as empty interfaces (lines 69 and 77), the mapping between the participants (lines 86, 118-120 and 130-132), an abstract pointcut to identify events in which observers are interested (line 140), and the code to notify (lines 148-153) and update (line 163) all observers afterwards.

To use the design pattern, a concrete observer instance, like `ColorObserver` in the example implementation, must extend and concretize the `ObserverProtocol`. The `ColorObserver` assigns the roles of the participants, provides a concrete definition of the events of interest and defines how the `Observers` should be updated on notification. Also see Figure 4.1.4. The roles are assigned by declaring the parent role interfaces. The events are captured by setting the abstract pointcut `subjectChange(Subject s)` to join points that change the color of an item.

```
35  public aspect ColorObserver extends ObserverProtocol{
..
..      /* Assings the Subject role to the Point class. */
42      declare parents: Point implements Subject;
..
..      /* Assings the Observer role to the Screen class. */
49      declare parents: Screen implements Observer;
..
..      /* Specifies the joinpoints that constitute a change. */
59      protected pointcut subjectChange(Subject s):
..          call(void Point.setColor(Color)) && target(s);
..
..      /* Defines how each Observer is to be updated on notification. */
69      protected void updateObserver(Subject s, Observer o) {
70          ((Screen)o).display("Screen updated because color changed.");
71      }
72  }
```

Figure 4.1.4: The concrete aspect `ColorObserver` extends the `ObserverProtocol` (line 35), which was partly shown in Figure 4.1.3. It assigns the `Observer` and `Subject` roles (lines 42 and 49), specifies the join points that are of interest (line 59) and concretizes the `updateObserver` method in order to update `Observers` (lines 69-71).

The role interfaces are only used as marker classes so that the mapping can take place in the concrete aspects. No additional code is introduced in the participants. The mapping between observers and subjects in maintained in the concrete pattern aspect, instead of in the subjects, as are the methods to update the observers. The actual mapping between observers and subjects can be done anywhere in the program. This way subjects and observers are optimally decoupled. The pattern can be reused easily, because the participants do not have to be aware of their roles.

4.1.5 Comparison of the Java and the AspectJ solution

Although the principle of a subject that notifies all of its observers on certain events remains the same, the solution structure of the Java and AspectJ implementation slightly changed. In both cases the roles are determined by the interfaces, albeit that the AspectJ implementation introduces the roles into existing components. A difference in structure is that the AspectJ implementation keeps the mapping between the participants and the method to notify observers in a central place, whereas they were respectively kept in the subject and in the observer with the Java implementation.

The AspectJ implementation is an important improvement over the Java implementation in two ways. Firstly, the superimposed roles of both the subject and the observers are modularized in the AspectJ solution. Program components do not have to known about their roles in advance. This way new or changing dependencies between objects can easily be added to existing programs. Secondly, the change in structure makes the actual dependency more modular, in that the event that triggers and update and the code that performs the actual update are kept in a central place. This eases development, maintenance and reuse of code

4.2 Composite

4.2.1 Intent

GoF: “Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly [1]”.

4.2.2 Recurring problem

Object-oriented decomposition often results in part-whole hierarchies of objects, which must then be composed or grouped somehow. An example of such a situation is a `Graphic` that is built up of `Points`, `Lines`, and possible other types of objects. Code that uses classes of a part-whole hierarchy must treat primitive and container classes differently, which makes programming complex. One would like to treat all objects in an identical way.

4.2.3 Structure of the Java solution

The composite design pattern uses a shared interface for both primitive classes and their containers. The interface assigns a single role to all different classes, namely that they are a *component* to be used in a composite data structure. The interface lists several methods that all components share, such as operations for accessing and managing their children. Clients use the shared interface to interact with all different kinds of classes in the composite structure in an identical way.

The shared interface `Component` contains prototypes of methods to add or remove children, to retrieve a certain child indexed by its number, and optionally of methods that are shared among all different components. The prototypes for the methods to add or remove a child to or from a `Component` respectively are `add(Component component)` and `remove(Component component)`. The prototype for the method to retrieve a child by its index is `getChild(int index)`. As every class that is to be used in the composite data structure must implement the interface `Component`, it is guaranteed that all objects can be treated in an identical way. The interface `Component` is shown in Figure 4.2.1.

```
38  public interface Component {
..
..      /* Methods to built a composite structure. */
46      public void add(Component component);
..
54      public void remove(Component component);
..
..      /* Methods to retrieve childs in the structure. */
62      public Component getChild(int index);
..
70      public int getChildCount();
71  }
```

Figure 4.2.1: The interface `Component` that should be implemented by all classes that are to be used in a composite data structure. It lists the prototypes to add or remove child `Component`s (lines 46 and 54), to retrieve a child `Component` by index (line 62) and to get the number of child `Component`s (line 70).

In the Java example solution two different classes, `CompositeA` and `LeafB`, implement the interface `Component`. In the main program, a composite data structure is built up with these `Component`s, after which it is demonstrated how each object in the structure can be treated identically. The method `toString()` is implicitly used as an example of a method that is shared among all different components.

4.2.4 Structure of the AspectJ solution

With AspectJ, the composite design pattern could be modularized into a reusable abstract aspect `CompositeProtocol`. Similar to the abstract observer aspect that was shown in Figure 4.1.3, the protocol includes the roles that are assigned by the composite pattern as empty interfaces. In the AspectJ implementation there is a super role `Component` and there are two sub roles `Composite` and `Leaf`, that respectively are `Component`s that can have and that can not have contain children. Furthermore, the protocol contains the logic for accessing and managing children. This is now done in a central place in the aspect, in contrast to the Java solution where each component takes care of the registration of its children.

In addition to the principles that are found in the Java composite, the abstract aspect `CompositeProtocol` also prepares for visitors to operate on the composite data structure. For this purpose the interface `Visitor` was defined for visitors that optionally may be implemented in a the concrete pattern instance. The method-forwarding logic that is needed by the visitor is included in the abstract aspect. The method `recurseOperation` is used for this purpose. If an operation is to be applied to the aggregate structure, each composite forwards it to its children. Because the method `recurseOperation` is protected, it can only be called from the concrete pattern instance. Some little modifications may allow visitors to visit the composite data structure directly from any place within the program. In Figure 4.2.2, the relevant part of the abstract aspect `CompositeProtocol` is shown.

```
...
139     protected interface Visitor {
147         public void doOperation(Component c);
148     }
...
...     /* Implements the method-forwarding logic for Composites. */
158     protected void recurseOperation(Component c, Visitor v) {
159         for (Enumeration enum=getAllChildren(c);enum.hasMoreElements();) {
160             Component child = (Component) enum.nextElement();
161             v.doOperation(child);
162         }
163     }
...
```

Figure 4.2.2: Part of the abstract aspect `CompositeProtocol` that prepares for visitors to operate on the composite data structure. Note that the method `recurseOperation` is protected (line 158) and thus can only be called from within the concrete pattern instance.

In the example, a concrete aspect `SampleComposite` is defined that does two things. First, it respectively assigns the roles `Composite` and `Leaf` to the existing classes `CompositeA` and `LeafB`, similar to the way this was done in Figure 4.1.4. Second, it introduces two client-accessible methods into the `Component`s. Both of these methods make use of `Visitor`s, for which the protocol includes the method-forwarding logic. For example, the method `printStructure(PrintStream s)` attaches an operation with arguments.

The actual composite structure is built in the main program, as in the Java implementation. There is however a difference in that child `Component`s are not added to a `Composite` object; instead the mapping between components and children is stored centrally in the pattern instance.

4.2.5 Comparison of the Java and the AspectJ solution

The Java and AspectJ solutions are similar in that each of the participants still uses the `Component` interface in order to be treated identically. An important difference is that the superimposed role of the `Component`s is modularized in the AspectJ solution. In the concrete

pattern instance, introduction is used to assign the `Composite` and `Leaf` roles and to attach methods that are shared among all participants.

The solution structure of the Java and AspectJ implementation is slightly different, because the mapping between components and their children is stored in the components itself with Java, whereas it is stored centrally in the concrete pattern instance with AspectJ.

The AspectJ implementation is an improvement over the Java implementation because of several reasons. Firstly, the participants do not have to know about their roles in advance; the composite design pattern thus can be used with existing classes. A direct result of this is gained flexibility in relation to the shared operations. The operations to be performed on the composite structure can easily be added or removed by modifying the concrete pattern instance with AspectJ, while they should be known in advance and listed in the `Component` interface with Java.

4.3 Visitor

4.3.1 Intent

GoF: "Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates [1]".

4.3.2 Recurring problem

For many applications, operations must be performed on data structures. Distributing operations across the various nodes of the data structure leads to systems that are hard to understand, maintain and change, because modifications to the operations must be done in the data structure itself. It would be better if new operations can be added separately, and if the node classes are independent of the operations that are applied to them.

4.3.3 Structure of the Java solution

The visitor design pattern encapsulates operations in visitor objects that can traverse the data structure. The visitor pattern assigns two roles, namely *visitor* and *element*. Modifications to the operations are done in the visitor objects, instead of in the data structure. New operations can simply be added by defining new visitors. The data structure on which the visitor objects work, however, must be prepared in advance to accept the visitor objects. The principle behind the visitor design pattern is a *double-dispatch* protocol: the visitor calls a method to be accepted in an element, after which the element is responsible for calling the appropriate method in the visitor that performs the operation.

Each visitor must implement the interface `NodeVisitor` that is shown in Figure 4.3.1, which tells the visitor what methods it should implement for visiting different the kinds of `Node`s. The example implementation contains the methods `visitRegularNode(Node n)` and `visitLeaf(Node n)` for respectively `RegularNode`s and `Leaf`s. The visit-methods contain the logic to perform the actual operations on the data structure.

```
35  public interface NodeVisitor {
..
43      public void visitRegularNode(Node node);
..
51      public void visitLeaf(Node node);
..
60  }
```

Figure 4.3.1: The interface `NodeVisitor` that must be implemented by each concrete visitor. The interface lists the visit methods for the various classes that can be used in the object structure to be visited. This `NodeVisitor` can visit `RegularNode`s (line 43) and `Leaf`s (line 51).

Each element of the data structure implements the interface `Node` that contains the method `accept(NodeVisitor v)`. When a `Node` is visited, it calls the visitor operation that corresponds to its class, as shown in Figure 4.3.2. The element supplies itself as an argument to this operation so that the visitor can access its state.

```
34  public class RegularNode implements Node {
..
..      /* The double-dispatch protocol in a RegularNode. */
54      public void accept(NodeVisitor nv) {
55          nv.visitRegularNode(this);
56    }
..
89  }
```

Figure 4.3.2: Demonstration how the double-dispatch protocol works in the class `RegularNode`. A `NodeVisitor` can be accepted as the interface Node is implemented (line 34). Once the the `accept` method is called (line 54), the `RegularNode` is responsible for the call-back to the appropriate visit method `visitRegularNode` in the the `NodeVisitor` (line 55).

The traversal strategy in the object structure is usually determined by the visitor class, but can be at other places as well. Traversal in the example implementation is done in the visitor. Once an element calls back the visitor, the visitor performs some operation and uses its knowledge about the element to forwards itself to the element's childs by calling their accept methods.

4.3.4 Structure of the AspectJ solution

With AspectJ, a part of the visitor design pattern could be abstracted into reusable code. The implementation uses an abstract aspect `VisitorProtocol`, shown in in Figure 4.3.3, that contains several empty interfaces to define the different roles assigned by the visitor pattern. These roles are `NodeVisitor` for concrete visitors and `VisitorNode` for concrete elements. To use the design pattern, a concrete visitor pattern instance `MyVisitor` is created that assigns the roles to the participants.

```
 51  public abstract aspect VisitorProtocol {
..
..      /* Interfaces for elements in the object structure. */
 58     public interface VisitorNode {}
..
 66     protected interface VRegularNode extends VisitorNode {}
..
 74     protected interface VLeaf extends VisitorNode {}
..
..      /* Interface to be implemented by concrete visitors. */
 80     public interface NodeVisitor {
..
 88         public void visitRegularNode(VisitorNode node);
..
 96         public void visitLeaf(VisitorNode node);
105     }
...
...     /* Attachment of double-dipatch protocol. */
113     public void VisitorNode.accept(NodeVisitor nv) {}
...
```

```
121       public void VRegularNode.accept(NodeVisitor nv) {
122           nv.visitRegularNode(this);
123       }
...
131       public void VLeaf.accept(NodeVisitor nv) {
132           nv.visitLeaf(this);
133       }
134   }
```

Figure 4.3.3: The abstract aspect VisitorProtocol that defines the roles of the participants and contains the code for the double-dispatch protocol. The elements to be visited (line 58) are either `VRegularNode` or `VLeaf` (lines 66 and 74); the visitor role is modeled by the interface `NodeVisitor` (lines 80-105). The `accept` methods with the appropriate call-back code (lines 133, 121-123 and 131-133) are automatically attached to the `VRegularNode`s and `VLeaf`s after the parent declaration in the concrete pattern instance.

As visitors are defined by their role, any class that implements the appropriate interface `VisitorProtocol.NodeVisitor` can be used as a concrete visitor. The different visit-methods that visitor classes must implement are listed in this interface. If needed, new visitor classes can simply be added afterwards. This is very similar to the Java solution where the only requirement also is that the visitor classes implement a shared interface. Another similarity is that the traversal strategy is implemented in each visitor class.

In contrast to the defining role that the visitors have, the `VisitorNode` role of the elements in the data structure is superimposed. The elements already have functionality and responsibility in the program. Two different kinds of concrete elements are distinguished, namely `VRegularNode`s and `VLeaf`s. In the concrete pattern instance `MyVisitor` all roles are assigned by means of a parents declaration of the classes `RegularNode` and `LeafNode`. As discussed in Subsection 1.2.2, this declaration automatically attaches the appropriate accept-method with the call-back functionality to each concrete element. The code that is responsible for this is shown in the bottom part of the `VisitorProtocol` in Figure 4.3.3.

4.3.5 Comparison of the Java and the AspectJ solution

The Java and AspectJ solution are similar in that they both rely on the fact that visitor objects implement a certain interface that lists the visit-methods that must be implemented. These visit-methods are in both cases called from the elements of the data structure. An important difference is that the superimposed role of the elements to be visited is modularized in the AspectJ solution. The elements do not have to be prepared in advance to accept visitor objects. The interface and the appropriate accept-methods are introduced in the concrete pattern instance.

The actual solution structure of the Java and AspectJ visitor design pattern did not change. Although the AspectJ implementation is more modular in terms of code, the resulting structure after declaring the parents of the data structure's elements is the same. Both solutions use separate visitors and rely on the *double-dispatch* protocol.

The AspectJ implementation is an improvement over the Java implementation, because of two reasons. Firstly, visitor objects can be used with existing data structures. Secondly, it is more modular as the code that is needed for all different concrete elements is maintained in a single place. This is eases development, maintenance and reuse of code.

An important observation is that only two different concrete element classes, namely `VRegularNode` and `VLeaf`, are distinguished in the `VisitorProtocol`. Although the abstract aspect works fine in the demonstration, it may not be reusable in realistic programs. In order to make the protocol more reusable, the different role interfaces should be moved to the concrete pattern instance. This modification pratically means taking out almost the entire reusable part of the current `VisitorProtocol`.

4.4 Proxy

4.4.1 Intent

GoF: “Provide a surrogate or placeholder for another object to control access to it [1]”.

4.4.2 Recurring problem

In [1], several situations in which a proxy can be used are distinguished. A *remote proxy* provides a local representative for an object in a different address space; a *virtual proxy* creates expensive objects on demand; a *protection proxy* controls access to the original object; and a *smart reference* is a replacement for a reference that performs additional functionality. A proxy thus acts as a stand-in for a real subject and forwards requests when appropriate, depending on the kind of proxy.

4.4.3 Structure of the Java solution

A Java proxy conforms to the interface of the subject it controls access to. This way the proxy acts just like the subject and the client that is using the proxy is unaware of the fact that it is not directly communicating with the subject. The proxy can deny or forward requests to the subject, depending on the type of proxy. The role that is assigned by the proxy design pattern can either be a defining or superimposed, depending on the whether the proxy has functionality or responsibility outside the pattern or not.

In the example Java implementation, the interface `Subject` defines how the subject behaves: concrete subjects should be able to `print` and `write` strings to `System.out`. Any class that functions as a proxy also should implement the interface `Subject`. For example, the class `Proxy` is set as a placeholder for `RealSubject` in the main program. Any requests to `Proxy` are simply forwared to `RealSubject`, while counting the number of calls to `print`. Although calls to `write` are not of interest for the proxy, the method still must be implemented in order to forward the request. See Figure 4.4.1 for the class `Proxy`.

```
34  public class Proxy implements Subject {
..
40      private Subject realSubject;
..
47      private int printCalls = 0;
..
..      /* @param the subject that is controlled. */
55      public Proxy(Subject subject) {
56          this.realSubject = subject;
57      }
..
66      public void print(String s) {
67          printCalls++;
68          realSubject.print(s);
69          System.out.println("[Proxy:] ... call "+printCalls+" ... ");
70      }
..
79      public void write(String s) {
80          realSubject.write(s);
81          System.out.println("[Proxy:] ... not of interested ... ");
82      }
83  }
```

Figure 4.4.1: The class `Proxy` can function as a surrogate for classes that implement the interface `Subject`. This interface lists the prototypes for the methods `print` and `write`. Although this proxy is only meant to count the number of calls to the method `print` (lines 66-70), the method `write` (line 79-81) must also be implemented to forwarded requests to the `Subject` (line 80).

4.4.4 Structure of the AspectJ solution

For the proxy design pattern an abstract aspect `ProxyProtocol` was defined that resembles the behavior of the *protection proxy* mentioned in Subsection 4.4.2. Similar to the other patterns that were discussed, the subject role is defined by means of an empty interface `Subject`. An abstract *pointcut* is used to capture all requests to the subject that should be protected. Accesses to protected parts of the subject are intercepted by means of an *advice declaration*; an abstract method `boolean rejection` is used to check whether the request can `proceed` or the method `handleFailedAccess` should be called. See Figure 4.4.2 for the abstract aspect `ProxyProtocol`.

```
 40  public abstract aspect ProxyProtocol {
 ..
 ..      /* Define the role to be assigned to Subjects. */
 46      protected interface Subject {}
 ..
 ..      /* Abstract requests that must be protected. */
 53      protected abstract pointcut protectedAccess();
 ..
 ..      /* Extend protectedAccess() pointcut for internal use. */
 60      private pointcut accessByCaller(Object caller):
 61          protectedAccess() && this(caller);
 ..
 ..      /* Intercepts accesses to protected parts of the Subject. */
 72      Object around(Object caller, Subject subject):
 73          accessByCaller(caller) && target(subject) {
 74
 75          if (! rejection(caller, subject, thisJoinPoint) )
 76              return proceed(caller, subject);
 77          return handleFailedAccess(caller, subject, thisJoinPoint);
 78      }
 ..
 ..      /* Checks whether the access should be denied or not. */
 90      protected abstract boolean rejection(Object caller,
 91                                           Subject subject,
 92                                           JoinPoint joinPoint);
 ..
 ..      /* Provides an alternative return value if access is denied. */
105      protected abstract Object handleFailedAccess(Object caller,
106                                                   Subject subject,
107                                                   JoinPoint joinPoint);
108  }
```

Figure 4.4.2: The aspect `ProxyProtocol` that abstracts a *protection proxy*. The abstract aspect defines the `Subject` role with an empty interface (line 53), provides an abstract pointcut `protectedAccess` to capture all requests to `Subject` that should be protected (lines 60 and 61) and declares what needs to be done with intercepted accesses (lines 72-78). Abstract methods are definded to check whether intercepted accesses should be rejected (lines 90-92) and to define alternative actions (lines 105-107) in case of rejection.

To use the `ProxyProtocol` it should be extended and concretized. Concrete pattern instances should define "which classes are subjects, which accesses should be protected, and what alternative value should be returned in case of a denied to a method [4]". The example AspectJ implementation provides two concrete proxies, namely `ProtectionProxy` and `DelegationProxy`. The former intercepts `print` calls from the class `Main`; the latter delegates `write` calls directed to the class `RealSubject` to `AnotherRealSubject`. In Figure 4.4.3, the `DelegationProxy` aspect is shown.

```
35  public aspect DelegationProxy extends ProxyProtocol {
..
41      declare parents: RealSubject implements Subject;
..
49      protected pointcut protectedAccess(): call(* RealSubject.write(..));
..
63      protected boolean rejection(Object caller,
64                                  Subject subject,
65                                  JoinPoint joinPoint) {
66          System.out.println("delegate all write() calls");
68          return true;
69      }
..
..      /* Alternative action is delegation to AnotherRealSubject. */
83      protected Object handleFailedAccess(Object caller,
84                                          Subject subject,
85                                          JoinPoint joinPoint) {
86          Object[] args = joinPoint.getArgs();
87          if (args != null)
88              AnotherRealSubject.write((String)args[0]);
89          else
90              AnotherRealSubject.write("");
92          return null;
93      }
94  }
```

Figure 4.4.3: A concrete pattern instance of `ProxyProtocol`. This `DelegationProxy` assigns the role `Subject` to the class `RealSubject` (line 41), tells that accesses to methods called `write` in that class are protected (line 49) and specifies the policy to be used for delegation by concretizing the methods `rejection` (lines 63-69) and `handleFailedAccess` (lines 83-93) that were declared abstract in the `ProxyProtocol`. The policy simply is to delegate all calls to `write` to `AnotherRealSubject`.

With the AspectJ implementation there is no need for an interface that lists the client accessible methods and that should be implemented by concrete subjects as well as proxies. The classes `RealSubject` and `AnotherRealSubject` thus do not have to know about their participation in the proxy design pattern. The methods that are of interest for any concrete proxy are directly identified by means of the concretized pointcut in the aspect.

4.4.5 Comparison of the Java and the AspectJ solution

The Java and AspectJ implementations of the proxy design pattern are quite different. With Java on the one side, the subject should be prepared in advance by implementing some interface. Proxies should implement all methods listed in the interface, even if they are not interested in them. With AspectJ on the other hand, no preparation has to be done in advance. The subject role is introduced by means of a parent declaration and the requests of interest are identified by means of the concretized pointcut. The additional abstract methods that needed to be concretized in the AspectJ example are the consequence of the fact that a specific type of proxy, namely a *protection proxy*, was implemented. For a other types of proxies other abstract methods may be specified in the proxy protocol.

The AspectJ implementation is an improvement over the Java version because of several reasons. First of all, the subject for which a proxy is needed does not have to be prepared in advance. At any time, a proxy can be added for existing classes. Secondly, proxies don't have to implement all possible requests, but can simply identify the ones that are of interest. Furthermore, the mechanisms and policies for, for example, a protection protocol are cleanly separated. The mechanisms to intercept calls and to verify whether they should be rejected are contained in the `ProxyProtocol`. The policy that tells what calls should be protected and when they should be blocked is in the `ProtectionProxy`. Separating mechanisms and policies increases the potential for reuse of code.

4.5 Decorator

4.5.1 Intent

GoF: “Attach additional responsibility to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality [1]”.

4.5.2 Recurring problem

Sometimes one wants to add extra responsibilities or functionalities to individual objects based on the context the object is in. Inheritance does not help in this case, because the add-ons are defined statically and apply to entire a class. With inheritance, a client cannot control how and when to decorate individual objects.

4.5.3 Structure of the Java solution

A Java decorator should conforms to the interface of the component it decorates so that its presence is transparent to the component’s clients. Requests from the client are forwarded to the component, while the decorator may perform additional actions. This is very similar to the proxy pattern that was discussed in Subsection 4.4. Multiple decorators can be composed dynamically, each forwarding the client’s request to the subsequent decorator. The decorator design pattern has two defining roles, *decorator* and *component*, and one superimposed role, *concrete component*.

The `Output` interface in the example implementation defines the role component. The only concrete component in the example is the class `ConcreteOutput`, which simply `prints` a string to `System.out`. Additional actions are added to `ConcreteOutput` by means of two decorators, `StarDecorator` and `BracketDecorator` that wrap other output around it. Each of the decorators extends the abstract helper class `OutputDecorator`, displayed in Figure 4.5.1. Note that `OutputDecorator` implements the interface `Output`, so that client requests can always forwarded to the component.

```
32  public abstract class OutputDecorator implements Output {
..
38      protected Output output;
..
47      public void print(String s) {
48          output.print(s);
49      }
..
..      /* @param output the component to decorate. */
60      public OutputDecorator(Output output) {
61          this.output = output;
62      }
63  }
```

Figure 4.5.1: The abstract helper class `OutputDecorator` which provides default implementation (lines 47-49 and 60-62) for each decorator class. Because this class implements the interface `Ouput`, client requests can always be forwarded to the component that is decorated.

As `OutputDecorator` provides a default implementation for the constructor, each of the actual decorators can simply call `super(…)` instead of setting the variable `Output output` themselves. All that has to be done is providing the code to decorate the requests that can be done by clients by overriding the method `print`. For example, in Figure 4.5.2 is shown how the `StarDecorator` adds three stars before and after the argument string before passing on the call to the component that is decorated.

```
41      public void print(String s) {
42          System.out.print(" *** ");
43          output.print(s);
44          System.out.print(" *** ");
45      }
```

Figure 4.5.2: The overriden method `print(String s)` in the `StarDecorator` decorator. Additional actions are performed just before (line 42) and just after (line 44) passing on the client's request to the component `output` that is decorated (line 43).

4.5.4 Structure of the AspectJ solution

In contrast to the other design patterns that were discussed in Subsections 4.5.1 to 4.5.4, no reusable aspect was defined for the decorator design pattern. Instead, the decorators are directly modeled as aspects that capture requests done by a client in order to perform additional actions before or after the request. The requests are captured by means of a *pointcut* and the additional actions to be performed are stated in an *advice declaration*. Because the execution points of interest will be the same for multiple decorators that are decorating the same concrete component, dynamic composition of decorators is no longer possible. Instead, the precedence of decorators must be explicitly defined in the aspects.

The AspectJ implementation of the decorator design pattern results in the same decoration as its Java equivalent. For this, two decorator aspects, `BracketDecorator` and `StarDecorator`, were defined. The latter decorator is shown in Figure 4.5.3. The keyword `dominates` means that `StarDecorator` has precedence over `BracketDecorator`. In the pointcut `printCall(String s)`, client requests to print a string are captured and the argument to be printed is extracted. In the *around* advice declaration, three stars are concatenated before and after the argument string before passing on the call to the component that is decorated.

```
32  public aspect StarDecorator dominates BracketDecorator {
..
..      /* Identifies the execution points of interest. */
39      protected pointcut printCall(String s):
40          call(public void ConcreteOutput.print(String)) && args(s);
..
..      /* @param s the string to be decorated. */
49      protected void around(String s): printCall(s) {
50          s = " *** " + s + " *** ";
51          proceed(s);
52      }
53  }
```

Figure 4.5.3: The aspect `StarDecorator` decorates a string to be printed with stars. It identifies the execution points of interest by means of a concrete pointcut (lines 39 and 40) and performs additional action in an advice declaration (lines 49 to 52). The keyword `dominates` (line 32) means that the `StarDecorator` has precedence over the `BracketDecorator`.

With the AspectJ implementation there is no need for an interface implemented by concrete components as well as decorators that lists the methods that the client can request. The class `ConcreteOutput` thus does not have to know about its role as a concrete component. The methods that are of interest for a decorator are directly mentioned in the pointcut of its aspect.

4.5.5 Comparison of the Java and the AspectJ solution

The Java and AspectJ solutions to perform additional functionality on client requests are rather different in structure. With the Java implementation on the one side, a shared interface is used so that decorators can wrap concrete components. Clients transparently deal with decorators that somehow extend functionality and forward request to concrete components. Additional responsibility can be attached to an object dynamically by wrapping decorators with other decorators. With the AspectJ implementation on the other hand, no shared interfaces are used. Decorators are implemented directly by means of an aspect that captures execution points of interest with pointcuts and then perform additional functions in an advice declaration. Decorators can no longer be dynamically composed, but the precedence must be explicitly defined in their aspects.

Compared to the Java implementation, the AspectJ implementation is an improvement because the components that must be decorated do not have to be prepared for this in advance. It is easily possible to add decorator aspects to existing classes or individual objects. Furthermore, similar to the proxy design pattern, Java decorators must implement all methods that are listed in the component interface even if they are not decorating certain methods, whereas AspectJ decorators only have to specify the additional responsibility for the execution points of interest.

The flexibility to dynamically compose decorators is lost with AspectJ, however, as the precedence of decorators must be set in advance. This also means decorator aspects cannot be reused as easily as that Java counterparts. For example, in order to change the order in which multiple decorators are applied, the decorator aspects must be modified instead of simply changing the order they are attached to each other.

5 Conclusions

Introduction

Design patterns are well-established in providing design expertise in programming, in particular in object-oriented programming (OOP). Design patterns are a way to share the general solution for design problems that are encountered over and over again during the development of software systems. Although design patterns are often mentioned in relation to OOP, they are also valuable in other programming paradigms, such as aspect-oriented programming (AOP).

Aspect-oriented programming (AOP) is an emerging programming paradigm based on the principle of modularization of crosscutting concerns. AOP languages are generally based on and provide extensions to OOP languages. For example, AspectJ is an aspect-oriented extension to Java. Existing object-oriented patterns possibly can be implemented more efficiently with AOP technologies or may be replaced by aspect-oriented patterns that provide a different solution structure for the same problem.

Related work

A survey of related work showed that existing object-oriented patterns have been implemented in several languages. In work by Peter Forbrig and Ralf Lämmel [5,6], the experimental pattern language *PaL* was used to create a reusable library covering of all 23 GoF patterns. It was shown by means of a simple example how *nested classes* and *reuse-clauses* could be used in order to program with patterns. The constructs that make reuse of design patterns possible are based on some kind of superposition of class structures. Classes can be merged with *PaL*'s reuse-clause to assign roles and introduce certain functionality.

More recent work by Jan Hannemann and Gregor Kiczales [3,4] implemented all GoF design patterns in both Java and AspectJ. For 12 out of the 23 GoF patterns, a substantial part of the implementation could be abstracted into reusable code, which resulted in a library with abstract pattern aspects. As AspectJ is intended to modularize crosscutting structures, most improvements were found for "patterns that superimpose behavior on their participants". A classification of patterns was made according to the usage of roles. Patterns that only assign *superimposed* roles benefit most from an aspect-oriented implementation, in contrast to patterns that assign *defining* or both kinds of roles. Important aspect-oriented language constructs that were used to implement the design patterns in AspectJ are *introduction* to assign roles and attach methods to the participants and *join points* to trigger certain events.

Results

In this bachelor project, the effects of AOP on existing object-oriented design patterns were studied. The main part of the project was devoted to a comparison of several Java and AspectJ pattern implementations to investigate whether the implementation structure changed or not and what benefits are obtained with AspectJ implementations. For this, the work of Jan Hannemann and Gregor Kiczales [3,4] was used as a starting point. A minor part of the time was spent to find out whether a correlation exists between the *pattern space* [1] and the benefits obtained with AspectJ. In the next few paragraphs, all results are presented.

	Observer	*Composite*	*Visitor*	*Proxy*	*Decorator*
Kinds of aspects	Concrete instance extends reusable abstract aspect.	Concrete instance extends reusable abstract aspect.	Concrete instance extends reusable abstract aspect.	Concrete instance extends reusable abstract aspect.	Decorator directly implemented by means of aspect.
Assignment of roles	Parent declaration for subject(s) and observers.	Parent declaration for composites and leafs.	Parent declaration for elements to be visited.	Parent declaration for concrete subjects of proxy.	-
Attachment functionality	-	Introduction of operations into components.	Introduction of accept methods into elements.	-	-
Event handling	Concretize an abstract pointcut to trigger events.	-	-	Concretize an abstract pointcut to trigger events.	Pointcut in aspect identifies execution points of interest.
Positive points	Use any existing classes. Change dependencies.	Use of existing component classes. Modify operations.	Use of existing element classes.	Use of existing subject classes. Separate policy.	Use of existing subject classes.
Negative points	-	-	Only two classes supported by pattern protocol.	-	No dynamic composition. Less reusable.
Room for improvemen	-	Directly accept visitor objects (see Appendix A).	More advanced child management (see Appendix A).	-	-

Table 5.1: An overview of the findings for the design patterns that were treated in Section 4. The first row shows the kinds of aspects involved in the AspectJ implementation. The next three rows show which of the constructs discussed in Subsection 1.2 were used in the implementation. The last three rows show the main benefits obtained with an AspectJ implementation as compared to Java, negative points that were found and possible improvements. The positive points "change dependencies" and "modify operations" for respectively the observer and composite pattern identify that it is easier to make modifications to existing code in AspectJ than in Java.

As mentioned above, a correlation between the kinds of roles a pattern assigns to its participants and how much the pattern implementation benefits from AspectJ was found in [3]. Therefore, a classification of patterns based on *superimposed* and *defining* roles was proposed. In the GoF book [1], another classification was presented, namely the *pattern space*, where patterns are classified according to their *purpose* and their *scope*. In Table 3.2, these properties have been compared with the benefits that are obtained with AspectJ. From the results in this table, it can be concluded that there is no correlation between the pattern space the benefits obtained with AspectJ.

In [3], design patterns were implementated in Java and AspectJ and it was found that the solution structure changed for some of the patterns, whereas it remained the same for other. In this project, the Java and AspectJ solution structure of five of the implemented patterns have been compared in detail to see how object-oriented pattern evolve in the presence of AOP. Both the solution structure and benefits obtained with AspectJ have been studied for the following patterns: observer, composite, visitor, decorator and proxy.

From the comparison of Java and AspectJ pattern implementations, the following conclusions can be drawn. The solution structure of some design patterns changed. AspectJ improved the modularity of the implementation and allowed design patterns to be used in combination with existing classes. An important observation is that AspectJ separates the *mechanism* and *policy* for some patterns (observer and proxy). This is an essential property for reuse. Because of the increased modularity, AspectJ implementations are easier to modify afterwards their Java counterparts (composite). For a few patterns, however, AspectJ introduced some negative points that are absent in the Java implementation (visitor and decorator). AspectJ thus is not always an improvement over Java. In Table 5.1, an overview that includes some of these findings is shown.

Theses

- In [3], a correlation between the kinds of roles that are assigned by a pattern and the benefits obtained with AspectJ was found. Patterns with only *superimposed* roles benefit most from AspectJ, because the roles crosscut the participant's classes.

- *Superposition* of class structures seems to be essential for programming with patterns. In [3] and [5,6], AspectJ's open class mechanism and *PaL*'s reuse-clause were respectively used for this.

- The aspect-oriented language constructs that allowed creating a library with several reusable design patterns in the form of abstract aspects are: *introduction* to assign roles and attach methods to the participants and *join points* to trigger certain events.

- Although AspectJ improves the modularity of many patterns, sometimes the AspectJ implementations have negative points that are not found in their Java counterparts. For example, dynamic composition is no longer possible with AspectJ decorators. Improved modularity thus does not always mean a better solution.

- AspectJ allows separating the *mechanism* and *policy* of some design patterns. If this is the case, the mechanisms are contained in an abstract aspect, whereas the policy is determined by a concrete pattern instance that contretizes all abstract elements.

Perspective

Some similarities and differences between the *PaL* and AspectJ approaches to create a reusable pattern library were already pointed out in this report, but further investigation may be needed. *PaL*'s concept of *reuse schemes* [5,6] was not treated here, but seems to be useful for the reuse of design patterns. Reuse schemes are very similar to generic abstract data types in C++, as they contain placeholders for the actual classes that are used with them. The Java programming language currently does not support generic types, but Sun Microsystems has proposed to extend Java in this direction. As such an extension directly affects AspectJ, investigation concerning 'reuse schemes' in AspectJ may be fruitful.

In this project the Java and AspectJ implementations of only 5 out of 23 GoF patterns were analyzed. As it turned out that AspectJ implementation is not always better than Java implementations, future analysis could cover the patterns that were not treated in this project. Furthermore, attention could be given to different implementation structures that are possible with AspectJ. In [3], it was claimed that more than one implementation structure was possible for some patterns, but each of the design patterns that were made available in [4] only contained a single AspectJ implementation. This raises the question what other AspectJ implementation structures exist and whether they are more useful or more efficient.

Appendix A: Performing operations on object structures

In this bachelor project a number of design pattern implementations was compared to find out how the implementation language affects the pattern structure. It was found that, compared to Java, a lot of design pattern implementations benefit from AspectJ. For some patterns, the AspectJ implementation had a different structure than the Java implementation. Work on the visitor design pattern by Joost Visser [11] and Jens Palsberg et. al. [10,11] showed that different solutions also could be obtained with the same implementation language. Some time of the project was devoted to finding out how different variants of the visitor pattern in Java can improve the solution for performing operations on object structures. After this, the variants were compared with AspectJ implementations of the visitor and composite design pattern that already have been discussed in this report.

In Subsection A.1, the principles behind the standard object-oriented visitor design pattern are introduced. In Subsection A.2, several variants of the visitor design pattern are shortly discussed. In Subsection A.3, the different visitor variants are compared to AspectJ implementations of the visitor and composite design pattern, and potential for improvement is mentioned.

A.1 The standard visitor pattern

In this subsection, the principles behind the standard object-oriented visitor pattern are discussed. Readers that are familiar with these principles can proceed to the variants that are described in Subsection A.2. For completeness, the intent of the visitor design pattern as defined by the Gang of Four [1] is: “Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates”.

The visitor design pattern deals with non-uniform data structures that consist of objects of many different classes. Each of the classes in the data structure can have distinct and unrelated operations. Directly implementing the logic to support an operation in the classes of the objects leads to tangled code. Another problem with this approach is that it is hard to apply new operations to existing data structures.

The visitor design pattern offers a solution for these problems by separating the logic of operations that must be performed on the data structure from the classes of the objects in the data structure. It puts the logic that is necessary to perform the operations in a separate visitor class, so that the classes of the objects in the data structure are not complicated by it. Because the logic is now separated from the classes, using different visitor classes can vary it.

During a traversal of the data structure one could have the visitor check, which object type it is currently dealing with and then apply the correct operation, but this leads to complex if-else statements. A more elegant solution for this problem is the double-dispatch protocol. For each object in the data structure, the visitor calls a method to be accepted in the object after which the object is responsible for calling the appropriate method in the visitor. This accept and call-back approach results in robust code, because it entirely replaces conditional statements.

The code that is responsible for navigating the data structure can be in the data structure itself, in the visitor class, or somewhere else in the program. In the first case the traversal code is fixed in the method that accept the visitor objects and traversal always happens in the same order. The second and third solutions are more flexible and keep the traversal code separate from the data structure. Usually the traversal is done within the visitor class, as the visitor has direct access to the elements of the data structure it is visiting.

A.2 Other ways to perform operations on object structures

Several possibilities for performing operations on object structures, such as the visitor design pattern, were already discussed during in this report. Some time was also spent to study two variations on the standard visitor pattern. These variants were not yet discussed. In this subsection, the possibilities for performing operations on object that were studied during this project are briefly introduced. The referenced work may be consulted for a more complete description of *visitor combinators* and *generic visitors*.

A.2.1 The visitor pattern in Java and AspectJ

The visitor pattern was part of the patterns that were studied in Section 4. The Java implementation of the visitor pattern is similar to the standard visitor that was introduced in the previous subsection. Subsections 4.3.3 and 4.4.4 respectively discusses the Java and AspectJ implementations of the visitor pattern. The comparison of both already showed that there were important differences between these two implementations. For example, the AspectJ implementation could be used with existing classes, but was limited with regard to the the identification of different class structures that concrete elements can have.

A.2.2 Visitor combinators

Work by Joost Visser [11] presented a variation on the visitor design pattern that allows composing new visitors from given ones and that obtains full traversal control. The standard visitor design pattern allows specialization of visitors, but it resists composition. Furthermore it limits the traversal control, because the traversal strategy is hard-coded in the accept-methods. To overcome these limitations, a small set of *visitor combinators* was proposed that can be used to construct new visitors from existing ones. A combinator is a reusable class that captures basic functionality and that can be composed with other combinators to obtain new functionality. Traversal combinators that rely on the notion of success and failure of visitors are used to control the traversal behavior. In Figure A.1 a generic formulation of the visitor combinator `One` is shown. In this formulation the abstract base class `AnyVisitor` with the method `visitAny` is a generic counterpart for a syntax dependent application.

```
 0  public class One extends AnyVisitor {
 1
 2      AnyVisitor v;
 3
 4      public void One(AnyVisitor v) {
 5          this.v = v;
 6      }
 7
 8      public void visitAny(Visitable x) throws VisitFailure {
 9          for (int i=0; i < x.nrOfKids(); i++) {
10              try {
11                  x.getKid(i).accept(this.v));
12                  return;
13              }
14              catch(VisitFailure f) { ; }
15          }
16          throw new VisitFailure();
17      }
18  }
```

Figure A.1: Generic formulation of visitor combinator `One` [11]. It encapsulates another Any`Visitor` passed in the constructor (lines 4-6) and tries to apply it to a `Visitable` exactly one time (lines 8-17). On success the method immediately returns (line 12); otherwise a `VisitFailure` is thrown (line 16).

A.2.3 Generic visitors

Work by Jens Palsberg et. al. [10,11] presented a variation on the visitor design pattern that circumvents the need for accept methods, meaning that elements of the data structure do not have to be prepared in order to accept a visitor. "This is achieved by using reflection to determine the fields of an object at run-time, and then visiting them all unless explicitly diverted by the visitor". These kinds of visitors are called *generic visitors*. Each generic visitor extends a base class `Walkabout` that contains all code dealing with reflection. The `Walkabout` base class only has a single visit-method that accepts an argument of the type `Object`. After accepting an `Object`, it uses reflection to determine the appropriate action to be performed. The pseudo-code for the `Walkabout` class is shown in Figure A.2.

```
class Walkabout {
    void visit(Object v) {
        if (v != null)
            if (this has a public visit method for the class of v)
                this.visit(v);
            else if (v is not of primitive type)
                for (each field f of v) this.visit(v.f);
        }
}
```

Figure A.2: Pseudo-code for the `Walkabout` class that has to be extended by each generic visitor [13]. The `Walkabout` base class contains all reflection code that determines the actions to be performed at each object.

A.2.4 The composite pattern in Java and AspectJ

The composite design pattern is somehow related to the visitor pattern. Although composite in the first place deals with composing several classes into tree structures and treating instances of different classes uniformly, the pattern also defines how operations on the elements of the composite structure can be performed. Especially the AspectJ implementation is interesting in this discussion, as it uses visitor objects within the concrete pattern instance to perform operations on composite structures. The composite pattern was already studied in Section 4. In Subsection 4.2.3 and Subsection 4.2.4, the Java and AspectJ implementations of the composite pattern were respectively discussed.

A.3 Comparison of ways to perform operations on object structures

Each of the ways to perform operations on object structures has different properties. Some of those properties are useful, whereas others are not. Because visitor combinators and generic visitors have not been desribed in detail, it is not possible to fully compare all techniques. In Table A.1, however, a selection of properties is shown for the various techniques that were discussed in the previous subsection. From the table it becomes clear that none of the techniques is perfect; which one is most useful depends on the context in which it is applied. In the table, several boxes that have interesting properties have been marked with a light-gray background.

By mixing the various techniques that have been discussed in Subsection A.2 it may be possible to create a new type of visitor that has a combination of the usefull properties of its predessors. Because the AspectJ visitor that was discussed in Subsection 4.3.3 has important limitations concerning the number of classes that is supported by the protocol, more research on different kinds of visitor implementations should be done. Unfortunately there was not enough time in this project to make a concrete proposal for an enhanced visitor.

	Java Composite	*AspectJ Composite*	*Java Visitor*	*Visitor Combinators*	*Generic Visitor*	*AspectJ Visitor*
Node classes independent of operations	No, nodes contain all operations.	Yes, operations introduced in concrete aspect.	Partly, nodes must comply to interface.	Partly, nodes must comply to interface.	Yes, class type determined by reflection.	Yes, interface introduced by concrete aspect.
Is it easy to add new operations	No, all node classes need modification.	Medium, add operations to concrete aspect.	Yes, define a new visitor for each operation.	Yes, define a new visitor for each operation.	Yes, define a new visitor for each operation.	Yes, define a new visitor for each operation.
Where is the responsibility for iteration	In operations defined in the structure.	In (abstract) aspect, called from operation.	In accept method of visitor class.	In various combinator classes.	In visitor and Walkabout base class.	In concrete visitor classes.
Control over the traversal strategy	Determined by existing operations.	Determined by introduced operations.	Some control in individual visitor classes.	Full control with traversal combinators.	Some control in individual visitor classes.	Some control in individual visitor classes.
Composition of operations	No, all nodes and interface must change.	Introduce new operation in concrete aspect.	No, only specilization of visitors.	Yes, with help of visitor combinators.	No, only specilization of visitors.	No, only specilization of visitors.
Use with existing data structures	No, structure built up with components.	No, structure built up with components.	Yes, if nodes implement interface.	Yes, if nodes implement interface.	Yes, if nodes implement interface.	Yes, interface is introduced into elements.
Is it easy to add types to data structure	Medium, if nodes comply to interface.	Medium, assign extra roles in concrete aspect.	No, modify visitor and node interface.	No, modify visitor and node interface.	Yes, visitors act on arbitrary structures.	No, two types are fixed in the visitor protocol.

Table A.1: Comparison of six different ways that allow performing operations on object structures. Boxes with interesting properties have been marked with a light-gray background. The Java and AspectJ composite and visitor design patterns were respectively discussed in Subection 4.2 and 4.3. Visitor combinators and generic visitors were shortly introduced in Subsection A.2.

References

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

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *LNCS*, pages 220–242, Jyväskylä, Finland, 9–13 June 1997. Springer-Verlag.

[3] J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. *Proceedings of OOPSLA 2002*. See http://www.cs.ubc.ca/~jan/AODPs/oopsla02-patterns.pdf.

[4] J. Hannemann and G. Kiczales. *Java and AspectJ implementation of GoF design patterns*. Download available from http://www.cs.ubc.ca/~jan/AODPs/gof.zip.

[5] Joint work with Stefan Bünnig, Peter Forbrig and Normen Seemann. A Programming Language for Design Patterns. *Proceedings of Informatik '99*, Reihe Informatik aktuell, Springer, 1999.

[6] P. Forbrig and R. Lämmel. Programming with Patterns. In *Proceedings TOOLS-USA 2000*. IEEE, 2000.

[7] Peter Forbrig, Ralf Lämmel and Danko Mannhaupt. Pattern-Oriented Development with Rational Rose. The Rational Edge, January 2001. Article available online at http://www.therationaledge.com/content/jan_01/t_rose_pf.html

[8] Special issue on aspect-oriented programming. *Communications of the ACM*, 44(10), October 2001.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, pages 327–353, 2001.

[10] J. N. Herder. Aspect-Oriented Programming with AspectJ. July 27, 2002. Paper created for OOP course at Vrije Universiteit. Online at http://www.cs.vu.nl/~jnherder/aop/paper.pdf.

[11] J. Visser. Visitor Combination and Traversal Control. *OOPSLA 2001 Conference Proceedings*, November 2001.

[12] J. Palsberg and C. B. Jay. The Essence of the Visitor Pattern. *Proc. 22nd IEEE Int. Computer Software and Applications Conf.*, COMPSAC, 1997.

[13] J. Palsberg, C. B.Jay and J. Noble. Experiments with Generic Visitors. 1998.

[14] The Java programming language. Java web site: http://java.sun.com.

[15] The AspectJ programming language. AspectJ web site: http://www.aspectj.org.