Disciplined Reuse of Aspects
State of the Art & Work Plan

André Restivo

September 4, 2007

Contents

List of Figures

Introduction

This document describes the work plan and state of the art for the PhD work of André Restivo started in 2006. Acceptance of this document by a steering committee is mandatory for the final registration in the Doctoral Programme in Informatics Engineering (ProDEI) at the Engineering Faculty of University of Porto.

Chapter 1 contains a brief description of the fundamentals of Aspect-Oriented Programming. It is intended for readers who are not acquainted with the main ideas behind this new software development approach.

Chapter 2 introduces the problem of conflicts and interferences between aspects and gives a brief idea of the current State of the Art concerning this particular problem.

Chapter 3 contains the thesis proposal, a description of the work done so far, and a work plan.

Appendix A lists some of the available research resources that might be helpful to guide this work to a successful ending. These resources include research groups and people working in the area and also important conferences and publications.

Chapter 1

Aspect-Oriented Programming

1.1 Separation of Concerns

Software Engineering (SE) is concerned with the theories, methods and tools needed to develop software with quality. The main goals of SE are to achieve better maintanability, dependability, efficiency and usability in software applications [1, 2]. Some of these goals can be attained by increasing software modularity, thus increasing code reusability, with obvious advantages. Modularity is all about keeping different concerns separated in contained modules. This is sometimes referred to as achieving a better *Separation of Concerns* (SoC). SoC is also, and according to Dijkstra [3, 4], "the only available technique for effective ordering of one's thoughts".

Through the years, software engineers have developed a reasonable quantity of programming paradigms and approaches. If analyzed in the correct perspective, all of these developments were thought to help developers achieve a better SoC. For instance, *Procedural-Oriented Programming* (POP) separates concerns into different procedures, *Object-Oriented Programming* (OOP) into classes and objects and the *Model-View-Controller* approach (MVC) separates content from presentation and logic.

Aspect-Oriented Programming, on the other hand, separates concerns into units of modularity called aspects. In the remaining of this chapter, this new development approach is presented and explained.

1.2 Object-Oriented Programming Issues

OOP is a programming strategy based on encapsulation and information hiding. It differs from the procedural approach in that it views a software system as a set of objects, with their own private state and behavior, rather than a set of functions sharing a global state. Object-oriented systems are easier to maintain as objects are independent and can be understood as standalone entities [5, 1]. The advent of OOP has been an enormous step in the right direction making the possibility of achieving a perfect SoC a real possibility. An important set of principles for OOP has long been established in literature [6]:

Separation of Concerns (SoC). Every important issue (or concern) should be considered in isolation [3, 4].

Low coupling. Every module should communicate with as few others as possible [7].

Weak coupling. If two modules communicate at all, they should exchange as little information as possible [7].

Information Hiding. All information about a component should be private to a component unless it is specifically declared public [8].

Logical Cohesion. Related components should be grouped together [7].

The Open Closed Principle (OCP). A module should be open for extension but closed for modification [9].

The Liskov Substitution Principle (LSP). A derived class may substitute a base class [10].

The Dependency Inversion Principle (DIP). High-level modules should not depend upon low-level modules. Both should depend upon abstractions [11].

Stable Dependencies Principle (SDP). A component should only depend upon components that are more stable than it is [12].

The Interface Segregation Principle (ISP). Many client specific interfaces are better than one general purpose interface [13].

The Law of Demeter (LoD). Each unit should have knowledge only about closely related units. The Law of Demeter is a design guideline for developing software, particularly OO programs. This law states that "*objects should talk only to their immediate friends*" [14, 15].

Even using OOP and other SE techniques, some concerns are, however, still inevitably tangled in the source code [16]. These concerns are normally referred to as *crosscutting concerns,* as they can spread throughout an entire software system. A common example of such a concern is the logging module of an application [17].

The inevitable tangling of crosscutting concerns happens because OOP is a single abstraction paradigm, meaning that we are forced into organizing the code following a single perspective (often referred to as the Tyranny of the Dominant Decomposition [18]). The *core concerns* of the application force other concerns, often non-functional, to get scattered throughout the application code. This tangling of concerns is one of the major contributors to the increased complexity of large software applications.

The same principles enunciated in the beginning of this section can also be applied to AOP, with AOP being an important step towards a better application of them [19].

1.3 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a recent software development approach which main objective is the encapsulation of *crosscutting concerns* in their own units of modularity, known as *aspects.* Aspects are later weaved back with the base modules at compilation, loading or execution time. Among others, AOP has the following main advantages [20]:

Explicitness. Crosscutting concerns are explicitly captured by aspects.

Reusability. It is possible through a single aspect to describe crosscutting concerns common to several components.

Modularity. Since aspects are modular units that encapsulate crosscutting structure and behaviour, AOP improves the overall modularity of an application.

Evolution. Evolution becomes easier since implementation changes of crosscutting concerns occur locally within an aspect and save the need to adapt existing classes.

Stability. Special AOP language support makes it possible to express generic aspects, which will remain applicable throughout future class evolution.

Pluggability. Since aspects are modular, they can be easily plugged in and out of an application.

1.4 AOP using AspectJ

To better understand how AOP works, some concepts must be explained first. *Joinpoints*, *pointcut designators*, *advices* and *aspects* are the pillars of most AOP languages. In the next sections these concepts are explained using the syntax of *AspectJ*, an AOP language based in Java and the current *de facto* standard for AOP [21].

1.4.1 Joinpoints

In order to specify where units and aspects should be weaved together, a set of points in the code where weaving can occur must be defined. These points are called *joinpoints*. Each AOP language may define its own set of possible joinpoints, normally referred as the joinpoint model for the language. AspectJ defines the following types of joinpoints:

- Method call or execution - *call(void Foo.doSomething(int))* or *execution(void Foo.doSomething(int))*;
- Constructor call and execution - *call(Foo.new())* or *execution(Foo.new())*;
- Read/write access to a field - *get(int Foo.myField)* or *set(int Foo.myField)*;
- Exception handler execution - *handler(FooException)*;
- Object and class initialization execution - *initialization(Foo.new())*.

This means that we can define a joinpoint as all points in the application where method *doSomething* of class *Foo* is called, or as all points where field *myField* of class *Foo* is changed. It is important to notice that many more possible joinpoints could have been defined but for one reason or another they have been left out of the AspectJ specification. For example, *joinpoints* defined as references to code line numbers could also have been created, but this would be an extremely dangerous way of defining joinpoints as code changes would easily and unexpectedly alter the application behaviour.

1.4.2 Pointcut designators

As crosscutting concerns are normally scattered throughout the application, we often want to weave an aspect to several different joinpoints. For this purpose, AspectJ introduced the notion of *pointcut designators,* which allow the combination of joinpoints. A pointcut designator refers to several joinpoints that can be easily combined using the following operators:

- *a* && *b* - all joinpoints belonging to *a* and *b* (with *a* and *b* being conditions)
- *a* || *b* - all joinpoints belonging to *a* or *b*
- !*a* - all joinpoints not belonging to *a*

This type of pointcut definition is normally referred as *name-based crosscutting,* as opposed to *property-based crosscutting,* that allows the use of several wildcards thus making pointcut definition easier and more effective:

- Any returning type - *call(* Foo.method(int))*;
- Any method or any method starting by some string - *call(* Foo.get*())*;
- Any number of parameters - *call(int Foo.method(..))*;

Several other primitives have been defined in the *AspectJ* language, namely:

- Execution code defined in a certain type - *within(Foo)*;
- Execution code defined inside a method with a certain signature - *withincode(int Foo.method())*;
- Boolean expression evaluates as true - *if(booleanexpression);*
- Arguments are of a certain type or of the same type of a certain identifier - *args(int)* or *args(identifier).*

Pointcuts must be named by the developer and can take a number of parameters. The last primitive just described (*args*) is also useful to bind arguments from joinpoint definitions to pointcut parameters. Listing 1.1 shows some examples of pointcut definitions.

Listing 1.1: Pointcut examples

```
pointcut fooChanged():
        call(public void Foo.set*(..));

pointcut fooAccessedFromBar():
        call(public * Foo.*()) && within(Bar);

pointcut fooIdChanged(int i):
                call(public void Foo.setId(int)) && args(i);
```

1.4.3 Advices

In AspectJ, the element that defines how an application behaviour is changed in order to implement a certain aspect is an *advice*. Advices are nameless code blocks that execute implicitly whenever a certain joinpoint, belonging to the pointcut associated to the advice, is reached. AspectJ defines three types of advices:

- *Before advices* run before the actual joinpoint code is executed but cannot prevent the joinpoint from being executed;
- *After advices* run after the actual joinpoint code is executed. There are variations for normal exit from a joinpoint (*returning*) and exit due to an exception (*throwing*);
- *Around* advices have total control over the joinpoint execution. The joinpoint is only executed if the keyword *proceed* is invoked. Around advices must declare a return value identical to the one declared by the joinpoints that triggers it.

Pointcuts can capture the context from the associated joinpoints and pass it to the advices. The *args* operator described in the previous section is one of the ways to accomplish this. Other operators allow capturing other parts of the joinpoint context:

- Capturing the object being called - *target(callee)*;
- Capturing the object currently being executed - *this(self)*;

Combined with *call* and *execution* joinpoints, these two operators allow a great deal of control over the joinpoint context. For example, in a *call joinpoint*, the *target* operator allows access to the object being called while the *this* operator allows access to the object calling the method. On the other hand, in a *execution joinpoint*, *this* refers to the object being executed and not to the one that called the method.

Listing 1.2 shows an example of an advice capturing the context from a joinpoint. The first set of instructions defines a pointcut that is activated every time any public method from the *Bar* class is called. By adding the *Bar* field to the signature of the pointcut (line 1) and adding the *this(bar)* joinpoint to the list of those captured by this pointcut (line 2), we are giving the chance for an advice to access the context of this pointcut. The second set of instructions (line 4) is an advice that uses this pointcut and has access to the *Bar* object being called.

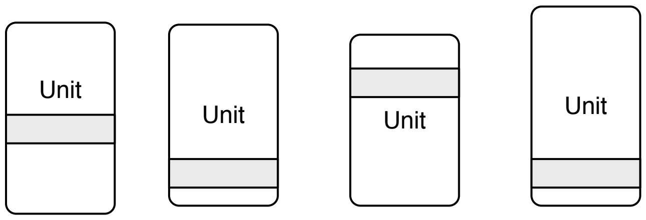

Figure 1.1: Scattered Cross-Cutting Concern

Listing 1.2: Advice example

```
pointcut fooAccessedFromBar(Bar bar):
        call(public * Foo.*()) && this(bar);

before() : fooAccessedFromBar(Bar bar) {
        // Do something to the Bar object that
        // called the method from class Foo
}
```

1.4.4 Aspects

Finally, the last element introduced by AspectJ is the *aspect*. An aspect combines pointcuts and advices composing a single modular unit that encapsulates a crosscutting concern. Aspects are very similar to classes as they are defined within packages, can be extended and can have attributes and methods. On the other hand, aspects cannot be instantiated using the *new* operator as classes are. Instead, aspects are automatically instantiated whenever a joinpoint associated to it is reached.

By default, aspects are *singletons*, meaning that there is only one instance of each aspect for the whole application. This behaviour can be altered with the use of the *perthis* or *pertarget* operators. These operators allow the existence of an instance for each calling object or for each target object. Another way of having more than one instance of the same aspect is by using the *percflow* operator. This operator creates an instance of the aspect for each flow of control of the joinpoints picked by the associated pointcut. Figure 1.1 shows a crosscutting concern scattered through several units while Figure 1.2 reveals the main AOP elements in action.

1.4.5 Inter-Type Declarations

Another powerful mechanism of the AspectJ framework are inter-type declarations (also known as *static introductions* or just *introductions*). This mechanism allows the introduction of new methods, fields or constructors into a class by an advice. The scope of the new class members is the aspect that introduced them

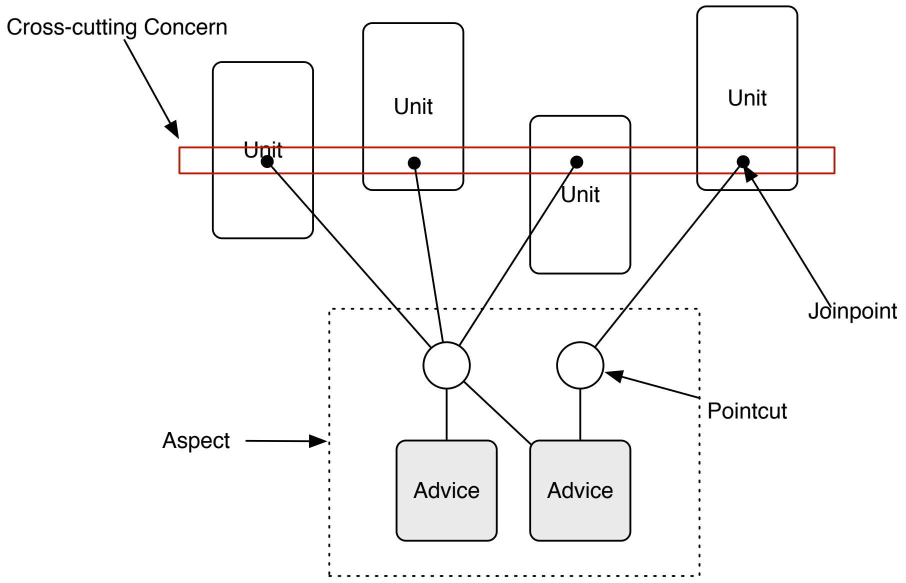

Figure 1.2: AspectJ key elements

and not the class where they were introduced. This makes it possible to define the new members as private to the aspect enforcing the modularity of aspects.

1.4.6 Declare Clauses

Some declarations about the advice can be added by developers with the use of the *declare* clause. For example, one could declare that a certain pointcut should never be reached using the *declare error* or *declare warning* clauses, allowing the implementation of system-wide policies. Other uses of the *declare* clause include allowing developers to change the class hierarchy. The *declare parents* clause can be used to make a class implement additional *interfaces* or to have a different *super-type*. The *declare precedence* clause allows developers to define the precedence between aspects.

1.4.7 Illustrative Example

Following is an illustrative example of some of the concepts just explained. The *Observer Pattern* is one of the most widely used *Design Patterns* defined by the GoF (name usually given to the authors of [22]). This pattern is used to define one-to-many dependencies between objects so that when one object is changed, all its dependents are notified automatically [22].

Listing 1.3 shows a simple way of implementing this pattern in a standard OOP way. The example chosen contains a class *Point* that notifies its observers (for example the *Screen* class) when changed. This implementation was not done exactly as defined by the *Observer Pattern,* for simplicity reasons.

Listing 1.3: Observer Pattern (OOP)

```
public class Point
{
        private float x;
        private float y;
        private Vector<Observer> observers = new Vector();

        Point (float x, float y)
        {
                this.x = x;
                this.y = y;
        }

        public void setX(float x){
                this.x = x;
                notifyObservers();
        }

        public void setY(float y){
                this.y = y;
                notifyObservers();
        }

        public void addObserver(Observer o){
                observers.add(o);
        }

        public void notifyObservers(){
                Iterator<Observer> itr = observers.iterator
                    ();
                while (itr.hasNext())
                        itr.next().subjectChanged(this);
        }
}
```

Although Design Patterns, and in particular the *Observer Pattern*, provide valuable design solutions for all developers, sometimes they originate other design problems as the one that can be identified by looking at the code at Listing 1.3. The *Point* class no longer represents a point, but instead it represents a point that notifies observers when changed. The representation of a point and the fact that points should notify dependant objects when their state changes are two different concerns that became inevitably tangled in the code.

Listing 1.4 shows the AOP version for the same implementation. In this case, the *Point* class is left untouched, while the *Observer Pattern* is completely contained in one unit of modularity. *Inter-type* declarations have been used to add an *observers Vector* to the *Point* class, as well as an *addObserver* method to the same class, and a *pointChanged* method to the *Screen* class. A *pointcut* has been defined as the execution of every method of the *Point* class starting with “set”. This pointcut also captures the *Point* object where the method was executed, from the current running context, to pass it to the associated advice.

A single advice has been created that simply iterates over the point observers (in this case screens) and calls the *pointChanged* method of each one of them. As this example shows, AOP can be extremely valuable in helping separating different concerns.

Listing 1.4: Observer Pattern (AOP)

```
public class Point
{
        private float x;
        private float y;

        Point (float x, float y)
        {
                this.x = x;
                this.y = y;
        }

        public void setX(float x){
                this.x = x;
        }

        public void setY(float y){
                this.y = y;
        }
}

public aspect PointObserver
{
        public Vector Point.observers = new Vector();

        private void Screen.pointChanged(){
                updateDisplay();
        }

        public void Point.addObserver(Screen s){
                observers.add(s);
        }

        protected pointcut pointChanged(Point p) :
                execution(void Point.set*(..)) && this(p);

        after(Point p) : pointChanged(p){
                Iterator itr = p.observers.iterator();
                while (itr.hasNext())
                        ((Screen)itr.next()).pointChanged();
        }
}
```

1.5 Aspect-Oriented Software Development

A large number of software development approaches exist. Nevertheless, most of them follow the same development lifecycle. Aspect-Oriented Software Development (AOSD), although not a software development methodology *per se*, can be used in each of the various activities of the development process. AOSD adds to the process with the inclusion of new tools and languages, but, at the same time, some activities must be changed in order to accommodate this new approach. For example, documentation techniques must evolve to cope with the new concepts and language constructs brought by AOP. The following sections summarize some of the existing approaches, each focusing a particular phase of the lifecycle.

1.5.1 Requirements Analysis

The requirements phase of the software development cycle is one that can benefit with the introduction of the AOP approach. Grundy [23] states that traditional requirements capturing techniques are not powerful enough to describe component requirements, leading to less reusable components. This study lead to a new requirements specification methodology named Aspect-Oriented Component Requirements Engineering (AOCRE).

Rashid [24] rationalized that SoC issues were a real problem even in the requirement analysis phase of the software development cycle, leading to another new requirement specification paradigm: Aspect-Oriented Requirements Engineering (AORE). Using aspects in this early phase of the process, Early Aspects [25], allow the identification of conflicts between crosscutting concerns earlier and at the same time it helps achieve a better traceability of system wide requirements throughout the development process. Using aspects in this phase also ensures better homogeneity in a aspect-oriented software development process. A new extension to the Unified Modeling Language (UML) notation has also been developed to support these new ideas [26].

Sutton [27] presented Cosmos, a software concern-space modeling schema. In this proposed schema, the author separates the notions of concerns, relationships and predicates. Concerns are categorized as logical and physical. Logical concerns are further typed as classifications, classes, instances, properties, and topics; physical concerns as collections. instances, and attributes. Relationships are categorized as categorical, interpretive, physical, and mapping. Predicates apply to concerns and relationships and reflect consistency considerations.

A promising approach has been introduced by Baniassad [28] and Clarke [29] with their *Theme Process*. This process is composed by two separate but related metodologies: Theme/Doc (that allows users to identify aspects in a set of requirements) and Theme/UML (how to model them in UML style designs).

Araújo [30] proposed an approach to model scenario-based requirements using aspect-oriented principles. His approach used Interaction Pattern Specifications (IPSs) to model aspectual scenarios. He also shown how these aspectual scenarios could be later composed with non-aspectual scenarios and transformed into executable state machines.

1.5.2 Design

The second phase of most software development methodologies is the high level design phase. The requirements captured in the Requirement Analysis phase are transformed into modules, classes and their interactions. Once again in this phase, the shift to AOSD brings some new challenges.

1.5.2.1 Specification

The most obvious problem found in this phase is how to adapt current architecture specification methodologies to cope with the new elements and ideas introduced by AOP. Several approaches to this problem exist, and not surprisingly most of them are based in the existing UML extension mechanisms, like the use of *stereotypes*.

Suzuki [31] states that crosscutting problems are very often only found in the construction phase of the development cycle. Developers usually deal with the problems found by adding aspects manually at that stage. This happens mainly due to the lack of aspect-oriented tools focused in this phase of the development cycle. The same author listed some of the advantages of incorporating aspects earlier, in the design phase:

Documentation and Learning. By visualizing aspects early in the design phase, developers can better understand how they interact in a more intuitive way. Also, this results in the early documentation of aspect usage.

Reuse of Aspects. The documentation of aspects in the design phase will allow the reuse of the aspectized components in different projects making it possible to create aspect libraries.

Round-trip Development. Incremental development is a common development strategy, where the various phases of the development process are repeated in order to fine tune any design flaws encountered during any of the phases. By adding aspects in the construction phase of the development process, developers are compromising the chance of going back to the design phase and change the system architecture.

In order to allow early aspect-oriented system designs, Suzuki [31] proposed extensions to the current UML diagrams supporting the design phase of the development process. In these new extensions, aspects would be represented as classes with a *aspect* stereotype. This is an obvious solution as aspects are much like classes, as they also have methods and attributes. The operation list compartment of the aspect would then show each weaving of the aspect as operations with the *weave* stereotype attached and a classifier to show which classes, methods and variables are affected by it.

There are three types of relationships possible between classes in the UML notation: Association, Generalization and Dependency. The same author states that, the kind of relationship between classes and aspects is better suited for a Dependency relationship, or, more precisely an Abstract Dependency relationship with a *realize* stereotype attached to it. Woven classes, the virtual classes that are the result of the weaving process, could then be represented simply as classes with the *woven class* stereotype.

Aldawud [32] has a similar, but more simplistic, approach using stereotypes to mark classes as being aspects, and associations between aspects and classes as having the *control* stereotype meaning that an aspect controls in some way that particular class.

Kande [33] proposed an alternative to this notation, with pointcuts being represented as new separate elements and adding an advice block to the aspect elements. New notations to represent AOP features, like the multiplicity of aspects, have also been proposed. The same author also presented some modifications to the collaborative diagrams of the UML in order to represent when joinpoints are reached.

Ho [34] has a different approach to the problem by using annotations and stereotypes as guides to the weaving process. For example, a class that should be made persistent could be marked as *persistent* and the weaving process would know which aspects had to be weaved in order to accomplish this.

Another different approach, using stereotypes to represent crosscutting concerns, advices and introductions (inter-type declarations), has been presented by Stein [35]. This work also proposes the use of UML interaction diagrams to represent joinpoints and collaboration diagrams to show how aspects interact with other units.

The Theme approach [29], already referred in the previous section, also allows the modeling of aspects in UML style diagrams.

A rather complete semantics for specifying pointcuts in UML diagrams has also been detailed in [36].

1.5.2.2 Design Patterns

Design Patterns are generic solutions to commonly recurrent problems. Patterns gained popularity in the software engineering community after the publication of the famous Design Patterns book by the GoF [22].

The advent of AOP is a great opportunity to redefine patterns in a more modular form. It has already been shown in Section 1.4.7 how one of those patterns, the *Observer Pattern*, could be applied using AOP.

Hannemann [37] has shown how Design Patterns could be mapped as aspects. The same author has described some of the advantages of using AOP to implement Design Patterns:

Locality. All code implementing the patterns is local to the pattern itself. None of the related classes are changed in the process.

Reusability. The pattern code can be reused throughout an application only by implementing a single concrete aspect (see Listing 1.5).

CompositionTransparency. If a class becomes involved in more than one Design Pattern (even in patterns of the same kind), each pattern can be reasoned about independently.

(Un)pluggability. Adding and removing patterns becomes as simple as removing the implementing aspect from the system.

Clarke and Kande [38, 39] have also written about how patterns could be implemented using AOP. Interestingly, both used the UML annotation extensions for AOP they proposed (see Section 1.5.2.1).

Garcia [40] has made an interesting study, comparing the implementations of all 23 Patterns proposed by the GoF in both OOP and AOP. This assessment study has shown that in many cases the AOP version of the Patterns provided a better SoC, better reusability and needed lesser number of lines of code.

1.5.3 Construction

The construction phase of the development cycle is when the actual code is written. Has AOP has several implication in this phase, several issues have been raised, namely: reusability, conflicts between aspects and accidental pointcuts.

1.5.3.1 Reusability

Achieving better reusability has always been one of the major goals of software engineering. The use of reusable modules reduces the implementation time and ensures that the used code has already been thoroughly tested and documented. As has been shown by Garcia [40], AOP allows developers to achieve better SoC in software applications, therefore better modularity and easier reusability is attained.

Listing 1.5 shows how the *Observer Pattern* can be implemented using AOP aiming for reusability. In this example, a generic *Observer Pattern* aspect was created. This aspect, created as an abstract aspect, specifies two new interfaces:

Observer and *Subject*. At this point these interfaces are not implemented by any class. Notice that this aspect is declared as being *perthis(subjectConstructed (Subject))*, meaning that one instance of this aspect exists for each *Subject* object. With this, one instance of the *observers* vector is created for each *Subject*. Then, the *addObserver* and *removeObserver* methods were created, as well as the abstract pointcut *subjectChanged*. An advice connected to this pointcut was also created. This advice will call the abstract method *updateObserver* for each *Observer* in the *observers* vector. Notice that the *subjectChanged* method captures the context in order to the advice to access the *observers* of the correct *Subject*. As this aspect never refers to the Screen or Point classes, it can be reused in the same software application or in other systems.

Binding this aspect to the correct classes is done by extending this aspect into a concrete aspect called *PointObserverPattern*. This aspect uses the declare clause to specify that the *Point* class implements the *Subject* interface and that the *Screen* class implements the *Observer* interface. This aspect defines also the concrete implementations of the *subjectChanged* pointcut and the *updateObserver* method making it possible to describe what events should be observed and what to do when these events occur. This example shows how suited AOP is to implement modular and reusable crosscutting concerns.

Listing 1.5: Reusable Observer Pattern (AOP)

```java
public abstract aspect ObserverPattern : pertarget(Subject s
    )
{
        public interface Observer { }
        public interface Subject { }

        Vector observers = new Vector();

        public void addObserver(Observer o)
        {
                observers.add(o);
        }

        abstract protected pointcut subjectChanged(Subject s
            );

        after(Subject p) : subjectChanged(p)
        {
                Iterator itr = observers.iterator();
                while (itr.hasNext())
                        ((Observer)itr.next).updateObserver
                            ();
        }

        public abstract void updateObserver(Observer o,
            Subject s);
}

public aspect PointObserverPattern extends ObserverPattern
{
        declare parents: Screen implements Observer;
        declare parents: Point implements Subject;

        pointcut subjectChanged(Subject s)
        {
                call(void Point.set*(..) && target(s));
        }

        public void updateObserver(Observer o, Subject s)
        {
                (Screen(o)).updateDisplay());
        }
}
```

Hanenberg [41] introduced some interesting rules on how to use aspects to achieve reusability:

- Separated pointcut declarations - Whenever a new aspect is created, a corresponding abstract super-aspect has to be implemented that contains all pointcuts declarations and definitions needed by the aspect. Advices in the sub-aspect refer to the pointcuts defined in the super-aspect.
- No pointcut for more than one advice - If one pointcut is used for more than one advice, there is no possibility to adapt the behavior of a single advice.
- Concrete aspects are always empty - Once an advice is within a concrete aspect, it becomes lost for any further reuse. In this way, a concrete aspect should not contain any pointcut definition (besides abstract ones). This guarantees the possibility to redefine the pointcuts in a concrete aspect.

1.5.3.2 Refactoring

Refactoring is the process of rewriting a computer program or other material to improve its structure or readability, while explicitly preserving its meaning and behavior. Several common methods for refactoring have been detailed in [42].

With AOP, new methods allowing the refactoring of existing code into this new paradigm have been described in [43, 44, 45]. Using these methods, it is possible to take an original OOP code and untangle it safely, turning it into AOP code. Unit tests can then be used to allow developers to refactor code making sure modules still behave correctly [46].

1.5.3.3 Debugging

Debugging has been an issue with AOP. When debugging AOP code, most frameworks use the result of the weaving process instead of the original aspects and class implementations. This may make debugging harder for AOP developers. Tools should improve by showing crosscutting structures, like thread trees that hide generated calls, and giving the ability to set breakpoints on pointcuts. [47] has an interesting approach both to the problem of debugging as well as to the related problem of profiling.

1.5.3.4 Development Environments

Before a new paradigm becomes ready for wide use and is accepted by industry development environments have to emerge. Several of these environments are already available for AOP. As AspectJ is currently the AOP leader language, it was expected that the first environments to appear would support this particular language. AspectJ is an extension of the Java language; Eclipse [48] is a successful open-source development environment, for that same language, that already had features like refactoring and unit testing. So, it was no surprise that the first good AOP development environments, AspectJ Development Tools (AJDT), appeared as extensions to the Eclipse IDE [49].

1.5.4 Testing and Validation

Testing and validation routines are important aspects of the Software Development cycle because no development paradigm ensures code correctness *per se*.

In AOP the main element is the aspect. Aspects differ significantly from classes and procedures. Testing AOP should not only test if the aspect performs as expected, but also if the classes modified by them continue to work correctly [50].

Zhao [50, 51] proposed three level of testing for aspect-based code:

Intra-module testing. Testing each individual element, such as advices methods and introductions.

Inter-module testing. Testing a public module along other modules it calls without considering invocations from other modules outside the aspect or class.

Intra-aspect testing. Testing the interaction between the aspect and multiple modules when they are called in a random sequence from outside the aspect.

1.5.5 Code Documentation

Code documentation is another important step of the development cycle. With the introduction of the new AOP elements, code documentation must be rethought.

AspectJ already offers AOP oriented documentation features such as the *ajdoc* documentation tool.

Besides AOP code documentation, another interesting possibility is to incorporate the AOP paradigm ideas in the code documentation process. In this way, documentation snippets could be thought as being documentation aspects that could be weaved together and, in this way, be composed into a complete document.

1.5.6 AOP Current Issues

Beyond the richness and potential of AOP, there are still several issues currently under heavy research. Most of them have already been introduced in the previous sections:

Specification. The current UML specification is not powerful enough to specify Aspect-Oriented systems. UML must be enhanced in order to incorporate the elements introduced by AOSD, either by using its extension mechanisms or by adding new features to the notation.

Conflicts. AOP promises greater component reusability but, on the other hand, aspects can be incompatible with each other which affects the idea of obliviousness.

Dependency. Aspects can depend on services provided by other aspects. A mechanism that allows aspects to specify which services are provided and required is needed to allow features like pluggable aspects.

Debugging. Debugging AOP code can be confusing because the program flow is not immediately easy to understand. Tools that help the debugging process must be further developed.

In the next chapter some of the issues that prevent better reusability and modularity of AOP code are addressed in more detail.

Chapter 2

Conflicts and Interferences

Besides providing higher software modularity, AOP also aims for a characteristic called *obliviousness* that states that developers should be able to implement application modules without any knowledge of previously implemented aspects or any future implementations [52, 53].

Current AOP languages, like AspectJ, are so powerful that *obliviousness* sometimes appears to be more of a problem than a solution. This is especially true when a large number of aspect modules are added into an application, often changing its behaviour, and thus making some aspect modules incompatible with each other. This happens because most aspects expect to be weaved into an application with a certain behavior and, if that behavior has been changed by a previously weaved aspect, then their own behavior could prove erroneous.

The next section presents a brief roundup of several classification terminologies found in literature that try to describe the different types of conflicts and interferences between aspects. Following this section, others describe prior work regarding how conflicts can be detected, solved and prevented.

2.1 The Anatomy of Aspect Interferences

Understanding a problem is always the first step to solve it. In this particular case it is important to identify the various kinds of interferences between aspects and how they emerge. Several researchers have tried to categorize aspect interaction according to different perspectives.

In the next sections three different approaches to aspect interference terminology and cataloging are presented. These approaches look at the same problem from different angles and each one of them has an important perspective into the problem.

2.1.1 The Interference Point of View

The most important work done in this area is probably the classification of aspect interferences by Tessier [54]. In his work the author points out several different ways in which aspects can interfere with each other:

Crosscutting Specifications - The use of joinpoints, and specially with '*' wildcards, can lead to accidental joinpoints or infinite recursions.

Aspect-Aspect Conflicts - When multiple aspects exist in the same system, problems like mutual exclusions between aspects, the importance of aspect ordering, or conditional execution of an aspect by another aspect can occur.

Base-Aspect Conflicts - Circular dependencies between aspects and basic classes.

Concern-Concern Conflicts - Aspects changing a functionality needed by other aspect and composition anomalies normally happening due to subtype substitutability.

2.1.2 The Aspect Point of View

According to Katz [55], three types of aspects can be described in respect to *how they affect an application.* This classification is important as some interferences only happen with some types of aspects. The three different aspect types are the following:

Spectative aspects only gather information about the system to which they are woven, usually by adding fields and methods, but do not influence the possible underlying computations;

Regulatory aspects change the flow of control (e.g., which methods are activated in which conditions) but do not change the computation done to existing fields;

Invasive aspects change values of existing fields (but still should not invalidate desirable properties).

2.1.3 The Dependency Point of View

Kienzle [56] approached the problem from a different point of view by considering only the *relationships of dependency between aspects and the original code.* Three different kinds of aspect dependencies have been identified:

Orthogonal aspects provide functionality to an application that is completely independent from the other functionalities of the application. No data structures are shared between these aspects and the rest of the application. This kind of aspects are very uncommon.

Uni-directional aspects depend from some functionality of the application. These can be further divided as *preserving*, if the application functionality is maintained or enhanced without any current functionalities being altered or hidden; or *modifying*, if the application functionality is altered or hidden.

Circular aspects are mutually dependent of each other. These kind of aspects are so tightly coupled that one can argue if they should really be considered as separate aspects or as one unique aspect.

2.2 Detecting Aspect Interferences

In order to solve the problem posed by the interference of aspects, a second problem must be solved first: how to detect that an aspect interferes with another aspect or module? Literature has many different ideas about how to solve this problem. In the following sections these ideas are explained.

2.2.1 Program Slicing

Balzarotti [57] claims that this problem can be solved by using a technique proposed in the early 80's called program slicing. A slice of a program is the set of statements which affect a given point in an executable program. According to the author the following holds:

> Let A1 and A2 be two aspects and S1 and S2 the corresponding backward slices obtained by using all the statements defined in A1 and A2 as slicing criteria. A1 does not interfere with A2 if $A1 \cap S2 = \emptyset$;

According to the author, this technique is accurate enough to identify all interferences introduced by an aspect but some of those are later considered to be false-positives (i.e. intentional interferences). Furthermore, the existence of pointcuts that are defined based on dynamic contexts, forces the analysis of every execution trace increasing the number of these false-positives. However the approach has the advantage of removing the burden of having to declare formally the expected behavior of each aspect.

2.2.2 Aspect Integration Contracts

Contracts have been introduced by Meyer [58] as a defensive solution against dependency problems in OOP. Some authors claim that contracts can be imported into the AOP world in order to assist programmers in avoiding interference problems.

Lagaisse [59] proposed an extension to the Design by Contract (DbC) paradigm by allowing aspects to define what they expect of the system and how they will change it. This will allow the detection of interferences by other aspects that were weaved before, as well as the detection of interferences by aspects that are bounded to be weaved later in the process. According to the author, for an Aspect A bound to a component C the following should be defined:

1. The aspect should specify what it requires from component C and possibly from other software components.
2. The aspect also needs to specify in which way it affects the component C and the functionality it provides (if applicable).
3. The specification of component C must express which interference is permitted from certain (types of) aspects.

This approach has the disadvantage of forcing the programmer to verbosely specify all requirements and modifications for each aspect as well as permitted interferences. On the other hand, the formal specification of behaviors has proven to be a valuable tool in Software Engineering.

2.2.3 Regression Testing

Katz [55] proposed the use of *regression testing* and *regression verification* as tools that could help identifying harmful aspects. The idea behind this technique is to use regression testing as normally and then weave each aspect into the system and rerun all regression tests to see if they still pass. If an error is found, either the error is corrected or the failing tests have to be replaced by new ones specific for that particular aspect.

2.2.4 Service-Based Approach

It has been noticed by Kienzle [56] that aspects can be defined as entities that require services from a system, provide new services to that same system and removes others. If there is some way of explicitly describing what services are required by each aspect it would be possible to detect interferences (for example, an aspect that removes a service needed by another aspect) and to choose better weaving orders.

2.2.5 Introduction and Hierarchical Changes Interferences

Störzer [60] developed a technique to detect interferences caused by two different, but related, properties of AOP languages. He claims that the possibility of aspects introducing members in other classes can lead to undesired behaviors as it can result in changes of dynamic lookup if the introduced method redefines a method of a superclass. He calls this type of interference *binding interference.*

The other problem Störzer refers to is the possibility of aspects changing the inheritance hierarchy of a set of classes. He claims that this type of changes can also give place to *binding interferences* as well as some unexpected behavior caused by the fact that *instanceof* predicates will no longer give the same results as before.

To detect this kind of conflicts the author proposes an analysis based on the lookup changes introduced by aspects.

Kessler [61] also studied how structural interferences could be detected. However, his approach is based in a logic engine where programmers can specify rules (ordering, visibility, dependencies, ...). He also described the different types of interferences that are possible with introductions and hierarchical changes and proposes solutions for each one of them.

2.2.6 Graph-Based Approach

Havinga [62] proposed a method based on modeling programs as graphs and aspect introductions as graph transformation rules. Using these two models it is then possible to detect conflicts caused by aspect introductions. Both graphs, representing programs, and transformation rules, representing introductions, can be automatically generated from source code.

Although interesting, this approach suffers the same problem of other automatic approaches to this problem as intentional interferences cannot be differentiated from unintentional ones.

2.3 Aspect Interference Resolution

Douence [63] [64] proposed a framework that allowed programmers to solve aspect interferences by using a dedicated composition language. The idea behind this language is to allow an explicit composition of aspects at the same execution point. The interferences solved by this approach are those that occur when the same crosscut is used by two different aspects.

2.4 Avoiding Aspect Interferences

The powerfulness of current AOP languages has been the target of several researchers that claim that without any control mechanisms, interferences will always be a big problem in the AOP world. In the next few sections some of the approaches that follow this path are described.

2.4.1 Robust Pointcuts

Recently, Braem [65] proposed a method based on Inductive Logic Programming in order to automatically discover intensional pattern-based pointcuts. This method aims at solving the *fragile pointcut problem*, that states that pointcuts defined by enumeration do not cope well with program evolution and that the use of wildcards to solve this problem can cause interferences by means of accidental joinpoints.

2.4.2 Crosscutting Interfaces

Crosscutting Programing Interfaces, or XPIs, have been introduced by Griswold [66] as a form of making AOP programming easier. By using abstract interfaces to expose pointcut designators, this approach decouples aspect code from the unstable details of advised code without compromising the expressiveness of existing AO languages or requiring new ones. The author expects that integrated-development-environment support could aid programmers by showing the scope of an XPI applicability.

2.4.3 Joinpoint Encapsulation

The reason why AOP is so powerful and at the same time so easily misused is that joinpoints are available for weaving without any knowledge of the programmer that originally developed the code. On one hand, this allows the developers to be oblivious about what code is going to be weaved in their code, but on the other hand is the source of interferences and conflicts. Larochelle [67] proposed the idea of adding joinpoint encapsulation by introducing a new kind of advice: join point encapsulation advice, or restriction advice. Restriction advice serves to encapsulate the join points selected by a pointcut against modification by other aspects thus enabling the modular representation of the encapsulation of crosscutting sets of join points.

Chapter 3

Thesis Proposal

In the previous chapters we have introduced AOP, a recent programming approach that promises a better SoC throughout all the development phases. The state of the art concerning AOP has been discussed with special emphasis on the problem of conflicts and interferences between aspects and base code.

Conflicts and interferences are one of the main issues preventing AOP from becoming a mainstream development approach. Obliviousness, the capability of developing aspects without having to reason about other concerns of the problem at the same time, has been tagged as the culprit of this problem.

Several ways of tackling this issue have been proposed in literature but none has proven sufficiently effective or even widely accepted by the community.

3.1 Objectives

Having in mind the difficulties of achieving modularity with Aspect-Oriented Programming, the proposed work has the following objectives:

1. To describe, formally, the different types of conflicts and interferences in order to better understand the scope of the problem.
2. To develop a test driven approach to allow the specification of incompatibilities and dependencies between aspects.
3. To develop a supporting tool, integrated into an existing development platform, to aid in the development of aspects by ensuring compatibility between them.

All these objectives aim to prosecute a more fundamental goal that is to improve the usability of AOP languages making obliviousness, when desired, a reality.

3.2 Thesis Statement

We believe that most – if not all – conflicts that arise between aspects, can be described as dependency problems. Aspects fail to interact correctly when behaviors that were expected to be implemented into the base system have been removed or altered by other aspects.

Behaviors can be specified, although incompletely, by tests in a test driven development. By allowing developers to specify which behaviors each aspect adds or removes and also those behaviors that each aspect requires it should be possible to detect conflicts and help developers determine what must be changed to correct such conflicts. Therefore, the thesis statement can be formulated as:

> A development approach (model, process and tools) enabling developers to specify the behaviors that each aspect adds, removes and requires would help on detecting possible conflicts between them.

This statement will be further detailed, explored, and validated during the doctoral work of this proposal.

3.3 Research Strategy

While the choice of the most adequate research method to use for software engineering is still a subject of debate, it is consensual that research methods can be grouped in four general categories: scientific, engineering, empirical and analytical [68, 69]. It is not in the scope of this document to discuss these methods, however, they will coexist in different phases of this work plan, and it can be briefly stated that:

- scientific methods will be applied for deriving theoretical models from real-world observations;
- engineering methods will be applied for developing concrete software solutions;
- empirical methods will be used mainly in the final part of the research to validate some parts of the achieved results.

3.4 Proposed Approach

Unit testing is used to informally prove the correctness of modules. Each module has its own set of unit tests. By running these tests one can verify if changes to a module have changed its tested external behavior. Unit tests can be seen

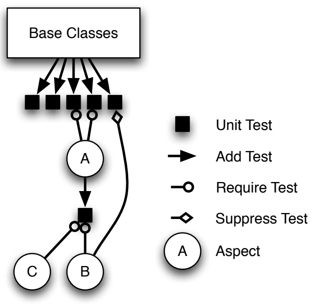

Figure 3.1: Aspects and Unit Tests

as a specification of the desired behavior of a module. With the introduction of aspects these specifications and their respective implementations can be easily changed by external entities, so units may no longer behave as expected. When an aspect is weaved into the code of an application, other aspects might have been weaved before and changed the expected behavior of the affected unit in a way that interferes with the new aspect being weaved.

In this way, it should be possible to specify which unit tests have to be valid for an aspect to be correctly weaved into the system. In the same way, it should be possible for an aspect to determine which tests it expects to break.

Many times aspects depend on each other. This happens when one aspect needs some behavior to be present in the system to work properly and this behavior is introduced by another aspect. It should also be possible for aspects to introduce new unit tests into the system specifying which new behaviors are being introduced by them.

It might also happen that an aspect needs a certain behavior to be present in the system but the unit providing this behavior does not have a specific unit test for this particular behavior. Aspects should be able to add new unit tests to code already in the application.

Figure 3.1 shows a possible notation illustrating an example where aspects add, remove and depend from unit tests. In this figure the dark square boxes are unit tests. The arrowed lines identify which unit or aspect created the unit test. The circled lines identify a dependency relationship, while lines with a diamond represent an invalidation of an unit test by an aspect (the big circles). From this explanation we can see that the base classes already provided several unit tests. Aspect "A" depends on two of those unit tests and adds another one. Aspect "B" depends on the unit test created by Aspect "A" and at the same time suppresses one of the initial unit tests. And finally, Aspect "C" also depends on the unit test created by Aspect A.

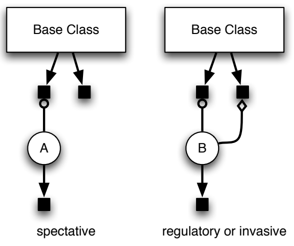

Figure 3.2: Different types of aspects

From this simple example we can already extract some conclusions: Aspect "A" is probably a *spectative* aspect (as defined by Katz) that simply added some new fields and methods to the unit; Aspect "B", on the other hand, has probably changed the behavior of the original code. We can also easily conjecture a possible order for the weaving process (e.g. "A" followed by "B" followed by "C").

Finding a possible weaving order in which dependencies between aspects are assured can probably be accomplished by using a simple Breadth-First Search (BFS) or using the A* algorithm (as this is a typical path finding in a graph problem). If such an ordering cannot be found then we are facing a conflict between aspects. In this case, an error message should be presented stating which aspects failed to weave, which unit tests are missing for these aspects, and which aspects removed them (if any).

The next section explains how this approach relates to Tessier and Katz classification of aspect interferences and conflicts.

3.4.1 Mapping aspects and interferences to unit tests

As we have seen in Section 2.1, aspects can be classified as being *spectative*, *regulatory* or *invasive*. Using the notation introduced in Section 3.4 we can depict these different type of aspects with relation to the unit tests they add, depend on, or suppress.

In Figure 3.2 there are two different aspects. Aspect "A" is probably a *spectative* aspect as it doesn't suppress any existing unit tests. It could also be an *invasive* aspect that happened to be "lucky" enough to change something the original developer wasn't expecting to be changed and didn't include in his unit tests. Aspect "B", on the other hand, is clearly a *regulatory* or *invasive* aspect as it suppresses some of the original unit tests that would fail after it had been weaved into the system.

In Figure 3.3, three types of interferences or conflicts are depicted. In the

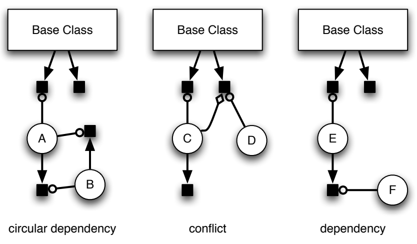

Figure 3.3: Different types of interferences

first one, aspects "A" and "B" are creating a circular dependency problem. The middle diagram depicts a conflict between two concerns, where aspect "C" is changing some functionality needed by aspect "D". The rightmost diagram shows aspects that need to be weaved in the correct order to function properly and at the same time a dependency between aspect "F" and "E".

This shows that if unit tests are correctly used they can help detecting most of the conflicts that aspects can introduce and have been plaguing AOP. The following sections show how these conflicts can be tackled with our proposed methodology with the help of a short example.

3.4.2 Using annotations to specify changes to unit tests

As has been stated before, breaking an unit test is not a clear sign of an aspect misbehaving. Due to their own nature, aspects are bound to change the functionality of other units of code and hence break their unit tests. In this way, aspects must have a way of announcing what unit tests they expect to break.

Very often, aspects are also depending on some functionality to be present into the system. This functionality can be delivered by the system base code or by other aspects. Conflicts between aspects are often caused by one aspect removing a functionality needed by another aspect, and dependency problems are commonly caused by one aspect expecting another aspect to deliver some functionality, which somehow is not effectively delivered.

Therefore, we claim that there is a clear need for aspects to be able to announce which aspects they are expected to break, which aspects they depend on, and which they are adding to the overall system. In this work we propose that aspects should be able to make this announcements using Java annotations. An example will now be introduced to explain how this could be attainable.

3.4.3 An example of conflicting aspects

Imagine a simple class depicting an User. This class would have fields like its username and password. It would also have setters and getters for those fields and a *verifyPassword* method. Listing 3.1 shows some simple unit tests that could have been used to ensure that the class was working properly.

Listing 3.1: User Class Unit Tests

```

public void testSetGetPassword() {
  user.setPassword("foo");
  assertEquals("foo", user.getPassword());
}

public void testVerifyPassword() {
  user.setPassword("foo");
  assertEquals(true, user.verifyPassword("foo"));
  assertEquals(false, user.verifyPassword("bar"));
}
```

It is also common that, for security reasons, passwords do not get stored in clear text. It is a common practice to store them using some hash function. However, to achieve a clear separation of concerns between the user data model and the security concern, this feature should be coded as a separate aspect. Listing 3.2 shows how this aspect could have been coded. By introducing these aspects some of the unit tests shown in Listing 3.1 get broken.

Listing 3.2: Encrypted Password Aspect

```

@SupressTest("user.UserTest.testSetGetPassword")
privileged aspect EncryptedPassword {
  protected pointcut
    passwordChanged(User user, String password):
    target(user) && args(password)
    && call(void setPassword(String));

  protected pointcut
    verifyPassword(User user, String password):
    target(user) && args(password)
      && call(boolean verifyPassword(String));

  void around(User user, String password)
    : passwordChanged(user, password)
  {
    // ... calculates md5 hash
    user.password =  md5hash;
  }

  boolean around(User user, String password)
    : verifyPassword(user, password)
  {
    // ... calculates md5 hash
    return (user.password.equals(md5hash);
  }
}
```

After introducing the aspect into the system, the developer should be warned that his aspect broke some unit tests. This could be easily computed by compiling and testing the system with and without the aspect. The developer could then inspect the broken unit test and decide if that would be an expected result from his aspect. In this case he would decide that it was because the getter and setter methods of the *User* class would not work as expected so he could just add a notation expressing that. The first line of Listing 3.2 shows how that notation could look like.

It is also common to prevent users from using passwords that are easily retrievable using brute force attacks. One way of doing it is to prevent them from using passwords that are too small. Once again, preventing this should be considered a separate aspect from the user data model and could be coded as seen in Listing 3.3. Notice that this aspect could have been coded in a much better fashion but for demonstration purposes it has been coded in a way that it needed the getter and setter methods of the original user class to work as originally intended. The developer should then announce that this aspect depends on the *testSetGetPassword* unit test. He could easily do so by adding a single line stating that in the beginning of the aspect.

Listing 3.3: Minimum Password Size Aspect

```

@RequiresTest("user.UserTest.testSetGetPassword")
@SupressesTest("user.UserTest.testVerifyPassword")
@AddsTest("user.UserTest.testVerifyPasswordML")
public aspect MinimumLengthPassword {
  protected pointcut
    setPassword(User user, String password)
    : target(user) && args(password)
      && call(void setPassword(String))
      && !within(MinimumLengthPassword);

  after(User user, String password)
  : setPassword(user, password)
  {
    if (user.getPassword().length()<6) {
      user.setPassword(password);
      throw new RuntimeException();
    }
  }
}
```

However, after introducing the aspect into the system, the developer would be warned that another aspect has suppressed that unit test. Besides that, this aspect would break the *testVerifyPassword* unit test. This is a typical case of a conflict between aspects. To solve this problem the aspect has to be rewritten in a different way and the broken unit test must be suppressed, and, perhaps, a new unit test should be added to verify if everything is still working.

This example shows how unit tests, if correctly used, can help detecting conflicts between aspects. It has also shown that the developer of each different concern did not have to know about other aspects being weaved into the system, at least until conflicts occurred, thus promoting obliviousness.

Figure 3.4 depicts the expected development flow when using our proposed approach. In this diagram we can see how desired behavioral modifications and accidental conflicts are treated in different ways. The diagram also shows how to reach a conclusion that the original requirements are the base of the conflict, in which case it would be impossible to correct the problem without changing them.

3.5 Work Plan

The plan devised for this research work encompasses seven main phases and will make use of diverse research methods and experimental approaches. These phases are:

1. State of the Art Review - This task aims at gathering information about the several topics relevant for the considered problem, namely: Aspect-Oriented Programming, Formal Methods (in particular Design by Contract) and Regression Testing (in particular Unit Testing).

2. Exploratory Projects Development - The development of one or several small projects using AOP in order to better understand the interference problems existing in the field and how they can be tackled.

3. Hypothesis and Problem Definition - The precise definition of the research questions and the formulation of a hypothesis will be the main result of this task. Furthermore, it comprises the definition of the parameters that will be used for validating and assessing the approach.

4. Conceptual Framework Development - Development of the conceptual framework that will allow the detection of interferences and conflicts between aspects.

5. Development of Supporting Tools - Development of a supporting tool, integrated into an existing development platform, to aid developing aspects using formal methods or regression testing to ensure compatibility between them.

6. Participation in Conferences:

   AOSD 2007 Paper already accepted and presented in the SPLAT workshop [70].

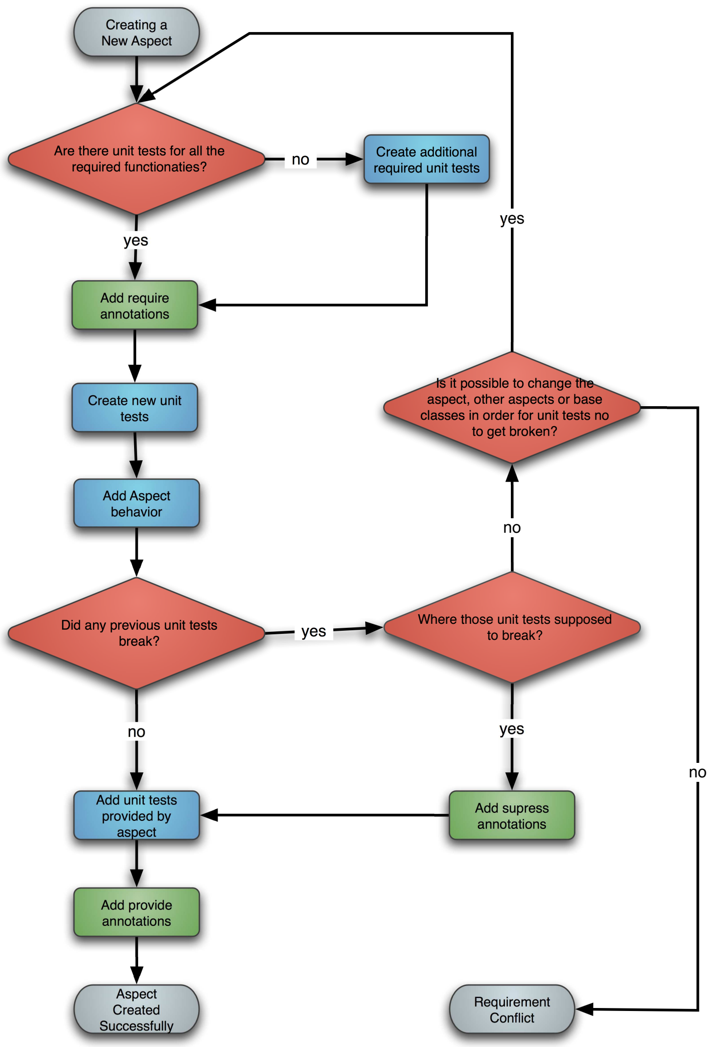
Creating a New Aspect
Are there unit tests for all the required functionaties?
no
Create additional required unit tests
yes
Add require annotations
Create new unit tests
Add Aspect behavior
Did any previous unit tests break?
yes
Where those unit tests supposed to break?
no
Is it possible to change the aspect, other aspects or base classes in order for unit tests no to get broken?
yes
no
yes
Add supress annotations
no
Add unit tests provided by aspect
Add provide annotations
Aspect Created Successfully
Requirement Conflict

Figure 3.4: Development flow for the proposed approach

OOPSLA 2007 Possibility of a paper describing further work developments. To be presented in a relevant workshop.

AOSD 2008 Possibility of a research paper describing the work done so far including some results. Probable deadline: September 2007.

OOPSLA 2008 Probable Deadline: March 2008.

TAOSD LNCS Transactions on Aspect-Oriented Software Development

7. Result Consolidation - This phase starts after the definition of validation criteria in the scope of the Hypothesis Definition, and gains momentum after a first version of the approach can be validated experimentally.
8. Writing of the Thesis - This task will accompany the considered phases for the research work as the respective milestones contribute for its completion.

Figure 3.5 shows the expected time frames for the execution of each of these tasks. Red milestones mark possible article submission deadlines to the conferences described in point 6.

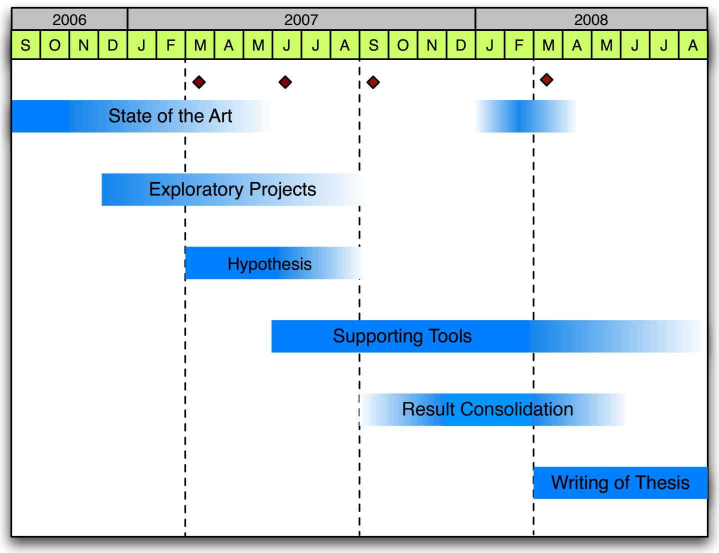

Figure 3.5: Work Plan

Appendix A

Resources

A.1 Research Groups and People

Following are listed some of the most prominent AOP and AOSD research groups, as well as some of the researchers currently working in AOP problems.

University of British Columbia

- **Gregor Kiczales** (http://www.cs.ubc.ca/~gregor/)

University of California

- **Cristina Lopes** (http://www.ics.uci.edu/~lopes/)

Twente Research and Education on Software Engineering (TRESE)

Software Engineering Group at the Department of Computer Science in the Faculty of Electrical Engineering, Mathematics and Computer Science at the University of Twente. Since 1988, the TRESE group has developed the aspect-oriented language Sina, which according to them was the first AOP language to be developed, and that later evolved into Composition Filters.

- **Mehmet Aksit** (http://wwwhome.cs.utwente.nl/~aksit/)
- **Lodewijk Bergmans** (http://wwwhome.cs.utwente.nl/~bergmans/)

AOSD Research Interest Group of Universidade Nova de Lisboa

Devoted to develop systematic processes and mechanisms for the use of aspect orientation from business modelling to implementation.

- **Ana Moreira** (http://ctp.di.fct.unl.pt/˜amm/)
- **João Araújo** (http://www-ctp.di.fct.unl.pt/˜ja/)
- **Miguel Pessoa Monteiro** (http://www-ctp.di.fct.unl.pt/˜mpm/)

Aspect-Oriented Software Engineering Group of Lancaster University

The Aspect-Oriented Software Engineering Group aims to develop systematic means for the identification, modularisation, representation and composition of crosscutting concerns (the aspects) throughout the software life cycle.

- **Awais Rashid**

Bedarra Research Labs

BRL is a private industrial research lab whose mission is to explore applications of next generation computing and communication technologies.

- **Brian Barry** (http://www.eclipsecon.org/2004/bios.htm#Barry)

IBM Research

- **Adrian Colyer** (http://www.aspectprogrammer.org/blogs/adrian/)
- **Harold Ossher** (http://www.research.ibm.com/people/o/ossher/)
- **Peri Tarr** (http://www.research.ibm.com/people/t/tarr/)

Technion - Israel Institute of Technology

- **Shmuel Katz** (http://www.cs.technion.ac.il/ katz/)

Concurrent Programming Research Group - Illinois Institute of Technology

- **Tzilla Elrad** (http://www.iit.edu/˜elrad/)

Research Institute for Advanced Computer Science

The Research Institute for Advanced Computer Science (RIACS) was created in 1983 under a cooperative agreement between the Universities Space Research Association (USRA) and the NASA Ames Research Center.

- **Robert E. Filman** (http://ic.arc.nasa.gov/people/filman/)

Demeter Research Team

A center dedicated to software related research and development in the areas of Adaptive, Object-Oriented, Aspect-Oriented, and Component-Based Programming. The Center is well known for the *Law of Demeter* and for an early definition of AOP without using the AOP terminology

- **Karl Lieberherr** (http://www.ccs.neu.edu/home/lieber/)
- **David H. Lorenz** (http://www.ccs.neu.edu/home/lorenz/)

Programming Principles and Practices Group

Programming Principles and Practices (PPP) is a group of researchers at Graduate School of Arts and Sciences, University of Tokyo.

- **Hidehiko Masuhara** (http://www.graco.c.u-tokyo.ac.jp/˜masuhara/)

Darmstadt University of Technology - Software Technology Group

- **Mira Mezini** (http://www.st.informatik.tu-darmstadt.de/staff/Mezini/)

University of British Columbia - Software Practices Lab

- **Gail Murphy** (http://www.cs.ubc.ca/˜murphy/)

Carnegie Mellon - Software Engineering Institute

- **Linda Northrop** (http://www.sei.cmu.edu/staff/lmn/)

Trinity College Dublin - Distributed Systems Group

- **Siobhán Clarke** (https://www.cs.tcd.ie/Siobhan.Clarke/)

University of Victoria - The MOD(ularity) Squad

- **Yvonne Coady** (http://www.cs.uvic.ca/˜ycoady/)

Concordia University - AOSD Interest Group

- **Constantinos Constantinides** (http://www.cs.concordia.ca/~cc/)

Ecole des Mines de Nantes - OBASCO Research Group

- **Remi Douence** (http://www.emn.fr/x-info/douence/)

A.2 Conferences

The most important conferences concerning AOP and AOSD are the following:

- Aspect-Oriented Software Development Conference (AOSD)
- Object-Oriented Programming Systems, Languages and Applications (OOPSLA)
- European Conference on Object-Oriented Programming (ECOOP)

Besides these conferences, there are others that share some of the ideas behind this work, like:

- Automated Software Engineering (ASE)
- Fundamental Approaches to Software Engineering (FASE)
- International Conference on Software Engineering (ICSE)
- European Software Engineering Conference (ESEC)
- Conference on Advanced Information Systems Engineering (CAISE)
- Conference on Model Driven Engineering Languages and Systems (Models)
- International Requirements Engineering conference (RE)

A.3 Journals

Some of the most important publications concerning AOP and AOSD are the following:

- LNCS Transactions on Aspect-Oriented Software Development
- IEEE Transactions on Software Engineering
- ACM Transactions on Software Engineering and Methodology

Bibliography

[1] Sommerville, I.: Software Engineering. 5th edn. Addison-Wesley (1995)

[2] Ross, D., Goodenough, J., Irvine, C.: Software engineering: Process, principles, and goals. Computer **8**(5) (May 1975) 17–27

[3] Dijkstra, E.W.: On the role of scientific thought (1974)

[4] Dijkstra, E.W. In: A Discipline of Programming. Prentice-Hall (1976)

[5] Dahl, O.J., Myhrhaug, B., Nygaard, K.: SIMULA 67 : common base language. Norsk Regnesentral (1968)

[6] Martin, R.C.: Design principles and design patterns. Technical report, Object Mentor (2000)

[7] Yourdon, E., Constantine, L.L.: Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice Hall (1979)

[8] Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12) (1972) 1053–1058

[9] Meyer, B.: Object Oriented Software Construction. Prentice Hall (1988)

[10] Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6) (1994) 1811–1841

[11] Martin, R.C.: The dependency inversion principle. C++ Report **8** (May 1996)

[12] Martin, R.C.: Stability. C++ Report (February 1997)

[13] Martin, R.C.: The interface segregation principle. C++ Report (August 1996)

[14] Lieberherr, K.J., Holland, I., Riel, A.J.: Object-oriented programming: An objective sense of style. In Meyrowitz, N.K., ed.: Proceedings of ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA 1988), San Diego, CA (September 1988) 323–334

[15] Lieberherr, K.J., Holland, I.: Assuring good style for object-oriented programs. IEEE - Software (September 1989) 38–48

[16] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: 11th Europeen Conf. Object-Oriented Programming. Volume 1241 of LNCS., Springer Verlag (1997) 220–242

[17] Harrison, W., Ossher, H., Tarr., P.: The beginnings of a graphical environment for subject-oriented programming. In Lopes, C., Mens, K., Tekinerdogan, B., Kiczales, G., eds.: Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'97. (1997) 65–66

[18] Tarr, P., Ossher, H., Harrison, W., Sutton, Jr., S.M.: N degrees of separation: Multi-dimensional separation of concerns. In: Proceedings of the 21st International Conference on Software Engineering (ICSE 1999), IEEE Computer Society Press (May 1999) 107 – 119

[19] Chavez, C., Lucena, C.: Guidelines for aspect-oriented design. In: First Brazilian Workshop on Aspect-Oriented Software Development (WASP) (SBES). (2004)

[20] Balzer, S., Eugster, P.T., Meyer, B.: Can Aspects Implement Contracts? In: Proceedings of RISE 2006 (Rapid Implementation of Engineering Techniques). (2006)

[21] AspectJ: AspectJ project home page (June 2006) http://www.eclipse.org/aspectj/.

[22] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts (1994)

[23] Grundy, J.: Aspect-oriented requirements engineering for component-based software systems. In: 4th IEEE International Symposium on Requirements Engineering, IEEE Computer Society (1999) 84–91

[24] Rashid, A., Moreira, A., Araújo, J.: Modularization and composition of aspectual requirements. In: Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD 2003). (2003)

[25] Rashid, A., Moreira, A., Araujo, J., Clements, P., Baniassad, E., Tekinerdogan, B.: Early aspects home page (2006) http://www.early-aspects.net/.

[26] Araújo, J., Moreira, A., Brito, I., Rashid, A.: Aspect-oriented requirements with UML. In Kandé, M., Aldawud, O., Booch, G., Harrison, B., eds.:

Second International Workshop on Aspect-Oriented Modeling with UML (UML 2002). (2002)

[27] Stanley M. Sutton, J., Rouvellou, I.: Modeling of software concerns in cosmos. In: AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development, New York, NY, USA, ACM Press (2002) 127–133

[28] Baniassad, E., Clarke, S.: Finding aspects in requirements with theme/-doc. In Tekinerdoğan, B., Moreira, A., Araújo, J., Clements, P., eds.: In Proceedings of Early Aspects 2004 Workshop. (March 2004)

[29] Clarke, S., Baniassad, E. In: Aspect-Oriented Analysis and Design: The Theme Approach. Addison Wesley Professional (October 2005)

[30] Araújo, J., Whittle, J., Kim, D.K.: Modeling and composing scenario-based requirements with aspects. In: Proceedings of the Requirements Engineering Conference, 12th IEEE International (RE'04), Washington, DC, USA, IEEE Computer Society (2004) 58–67

[31] Suzuki, J., Yamamoto, Y.: Extending UML with aspects: Aspect support in the design phase. In: ECOOP Workshops. (1999) 299–300

[32] Aldawud, O., Elrad, T., Bader, A.: A UML profile for aspect oriented modeling. In De Volder, K., Glandrup, M., Clarke, S., Filman, R., eds.: Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001). (2001)

[33] Kandé, M.M., Kienzle, J., Strohmeier, A.: From AOP to UML: Towards an Aspect-Oriented Architectural Modeling Approach. In: the Second International Workshop on Aspect-Oriented Modeling with UML, in conjunction with the Fifth International Conference on the Unified Modeling Language - the Language and its Applications (UML2002), September 30 - October 4, 2002, Dresden, Germany. (2002) Also available as Technical Report IC/2002/58, Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences.

[34] Ho, W.M., Pennaneac'h, F., Jézéquel, J.M., Plouzeau, N.: Aspect-oriented design with the UML. In Tarr, P., Finkelstein, A., Harrison, W., Nuseibeh, B., Ossher, H., Perry, D., eds.: Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000). (2000)

[35] Stein, D., Hanenberg, S., Unland, R.: Designing aspect-oriented crosscutting in UML. In Aldawud, O., Booch, G., Clarke, S., Elrad, T., Harrison,

B., Kandi, M., Strohmeier, A., eds.: Workshop on Aspect-Oriented Modeling with UML (AOSD-2002). (2002)

[36] Pawlak, R., Duchien, L., Florin, G., Legond-Aubry, F., Seinturier, L., Martelli, L.: A UML notation for aspect-oriented software design. In Aldawud, O., Booch, G., Clarke, S., Elrad, T., Harrison, B., Kandi, M., Strohmeier, A., eds.: Workshop on Aspect-Oriented Modeling with UML (AOSD-2002). (2002)

[37] Hannemann, J., Kiczales, G.: Design pattern implementation in Java and AspectJ. In: Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications, ACM Press (2002) 161–173

[38] Clarke, S., Walker, R.J.: Composition patterns: An approach to designing reusable aspects. In: Proc. 23rd Int'l Conf. Software Engineering (ICSE). (May 2001) 5–14

[39] Kande, M., Crettaz, V.: Towards patterns for concern-oriented software architecture. In Aldawud, O., Kandé, M., Booch, G., Harrison, B., Stein, D., Gray, J., Clarke, S., Santeon, A.Z., Tarr, P., Akkawi, F., eds.: Workshop on Aspect-Oriented Modeling with UML (AOSD-2003). (2003)

[40] Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., von Staa, A.: Modularizing design patterns with aspects: A quantitative study. In: AOSD 05. (2005) 3–14

[41] Hanenberg, S., Unland, R.: Using and reusing aspects in AspectJ. In De Volder, K., Glandrup, M., Clarke, S., Filman, R., eds.: Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001). (2001)

[42] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)

[43] Monteiro, M.P.: Refactorings to Evolve Object-Oriented Systems with Aspect-Oriented Concepts. PhD thesis, Departamento de Informática, Universidade do Minho, Portugal (2005)

[44] Monteiro, M.P., Fernandes, J.M.: The search for aspect-oriented refactorings must go on. In Tourwé, T., Kellens, A., Ceccato, M., Shepherd, D., eds.: Linking Aspect Technology and Evolution. (2005)

[45] Monteiro, M., Fernandes, J.M.: Towards a catalog of aspect-oriented refactorings. In Tarr, P., ed.: Proc. 4rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2005). (2005) 111–122

[46] Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools (The Addison-Wesley Object Technology Series). Addison-Wesley Professional (October 1999)

[47] Mehner, K., Rashid, A.: Towards a standard interface for runtime inspection in AOP environments. In Chu-Carroll, M.C., Murphy, G.C., Clarke, S., Estublier, J., Finkelstein, A., Harrison, B., Newman, E., eds.: Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2002). (2002)

[48] Eclipse: Eclipse home page (June 2006) http://www.eclipse.org/.

[49] Ajdt: Aspectj development tools (ajdt) home page (June 2006) http://www.eclipse.org/ajdt/.

[50] Zhao, J.: Tool support for unit testing of aspect-oriented software. In Chu-Carroll, M.C., Murphy, G.C., Clarke, S., Estublier, J., Finkelstein, A., Harrison, B., Newman, E., eds.: Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001). (2002)

[51] Zhao, J.: Unit testing for aspect-oriented programs. Technical Report SE-141-6, Information Processing Society of Japan (IPSJ) (May 2003)

[52] Filman, R., Friedman, D.: Aspect-oriented programming is quantification and obliviousness (2000)

[53] Filman, R.: What is aspect-oriented programming, revisited (2001)

[54] Tessier, F., Badri, M., Badri, L.: A model-based detection of conflicts between crosscutting concerns: Towards a formal approach. In: International Workshop on Aspect-Oriented Software Development. (2004)

[55] Katz, S.: Diagnosis of harmful aspects using regression verification (2004)

[56] Kienzle, J., Yu, Y., Xiong, J.: On composition and reuse of aspects. In: Software engineering Properties of Languages for Aspect Technologies. (2003)

[57] Balzarotti, D., Monga, M.: Using program slicing to analyze aspect-oriented composition (2004)

[58] Meyer, B.: Applying "design by contract". IEEE - Computer **25**(10) (1992) 40–51

[59] Lagaisse, B., Joosen, W., De Win, B.: Managing semantic interference with aspect integration contracts. In: Software Engineering Properties of Languages and Aspect Technologies. (2004)

[60] Störzer, M., Krinke, J.: Interference analysis for AspectJ. In: Foundations of Aspect-Oriented Languages (FOAL). (2003)

[61] Kessler, B., Tanter, É.: Analyzing interactions of structural aspects. ECOOP Workshop on Aspects, Dependencies and Interactions (ADI) (2006)

[62] Havinga, W., Nagy, I., Bergmans, L., Aksit, M.: A graph-based approach to modeling and detecting composition conflicts related to introductions. In: AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development, New York, NY, USA, ACM Press (2007) 85–95

[63] Douence, R., Fradet, P., Südholt, M.: Detection and resolution of aspect interactions. Technical Report RR-4435, INRIA (April 2002)

[64] Douence, R., Fradet, P., Südholt, M.: Composition, reuse and interaction analysis of stateful aspects. In: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD). (2004) 141–150

[65] Braem, M., Gybels, K., Kellens, A., Vanderperren, W.: Inducing evolution-robust pointcuts. ERCIM Evolution Workshop (2006)

[66] Griswold, W.G., Shonle, M., Sullivan, K., Song, Y., Tewari, N., Cai, Y., Rajan, H.: Modular Software Design with Crosscutting Interfaces. IEEE - Software **23**(1) (January/February 2006) 51–60

[67] Larochelle, D., Scheidt, K., Sullivan, K.: Join point encapsulation. In: Software Engineering Properties of Languages and Aspect Technologies. (2003)

[68] Tichy, W. F., H.N., Prechelt, L.: Summary of the dagstuhl workshop on future directions in software engineering. ACM SIGSOFT Software Engineering Notes **18**(1) (1993) 35–48

[69] Zelkowitz, M.V., Wallace, D.R.: Experimental models for validating technology. IEEE Computer **31**(5) (1998) 23–31

[70] Restivo, A., Aguiar, A.: Towards detecting and solving aspect conflicts and interferences using unit tests. In: Software Engineering Properties of Languages and Aspect Technologies. (2007)