Can Aspects Implement Contracts?

Stephanie Balzer⋆, Patrick Th. Eugster, and Bertrand Meyer

ETH Zurich (Swiss Federal Institute of Technology),
Department of Computer Science,
CH-8092 Zürich, Switzerland
{first_name.last_name}@inf.ethz.ch

Abstract. Design by Contract[TM] is commonly cited as an example of
the "crosscutting" concerns that aspect-oriented programming can ad-
dress. We test this conjecture by attempting to implement contracts
through aspects and assessing the outcome. The results of this experi-
ment cast doubt on the validity of the conjecture, showing that aspects
appear unable to capture contracts in a way that retains the benefits of
the Design by Contract methodology.

1 Introduction

Aspect-oriented programming (AOP) [1] emerged from a criticism of earlier pro-
gramming approaches, which it blames for focusing on just one dimension of
design. Object-oriented languages, for example, focus on the dimension of *data
abstraction* [2]. According to the AOP analysis, this prevents capturing other di-
mensions and results in tangled code, where code fragments representing dimen-
sions not supported by the programming language are replicated throughout the
code, defeating the advertised benefits of reuse and economy. Such dimensions,
or *aspects*, are said to "crosscut" the basic decomposition units of the software.
AOP instead supports multiple dimensions of design by turning such aspects
into first-class programming concepts, to be directly supported by the program-
ming language. Aspects, in the AOP view [3], allow programmers to capture
crosscutting concerns in a concise and modular way.

As an example of a crosscutting concern, the AOP literature frequently cites
[4, 5, 6, 7, 8] *Design by Contract*[TM](DbC)[1]. DbC [9] is a methodology to guide
many aspects of software development including in particular analysis, design,
documentation and testing. It is based on the specification of contracts that gov-
ern the cooperation between components of a system by specifying each party's
expectations and guarantees.

In the DbC view [9], contracts should appear in the components themselves.
Replacing contracts by aspect-based techniques would contradict this view since
aspects, by definition, appear separately from the program units they "crosscut".

⋆ Contact address: Stephanie Balzer, Chair of Software Engineering, Clau-
siusstrasse 59, 8092 Zürich, Switzerland.
[1] Design by Contract is a trademark of Eiffel Software.

We will start from this contradiction and investigate whether it can be resolved; for this purpose we will take the assertion that "aspects can emulate contracts" at face value and attempt to build such an emulation in practice.

The analysis uses a simple example that we first implement in *Eiffel* [10], the "mother tongue" of Design by Contract, and then in $AspectJ^{TM}$, a general-purpose aspect-oriented extension to Java and the most prominent realization to date of AOP ideas.

We illustrate through this example the loss of some of the principal benefits of Design by Contract when emulating contracts with aspects, in particular documentation and support for inheritance-based refinement. We further show that contracts are not crosscutting by nature, and that aspectizing them risks creating interactions with other aspects.

The remainder of this discussion is organized as follows. Section 2 provides a short introduction to AOP. Section 3 illustrates the concepts of DbC and introduces the running example of this paper. Section 4 studies, on the basis of the running example, how to aspectize contracts in AspectJ. Section 5 — the core of our comparative analysis — assesses whether and how this aspect-based emulation achieves the goals of the original contract-based design. Section 6 draws from this analysis to discuss the general question of whether aspects can emulate DbC. Section 7 provides our conclusion.

2   Aspect-Oriented Programming

This section provides an overview of *Aspect-oriented programming* (AOP) [1] and introduces its underlying terminology.

2.1   Overview

AOP promises improved separation of design concerns by modeling aspects separately from components. Typical applications of AOP include [11, 4] logging, concurrency/synchronization, change propagation, security checking, and — the claim examined in this paper — DbC enforcement. The announced benefits of aspect-oriented techniques include [4, 12, 13]:

- *Explicitness.* Aspects explicitly capture the structure of crosscutting concerns.
- *Reusability.* Since aspects can apply to multiple components, it is possible through a single aspect to describe crosscutting concerns common to several components.
- *Modularity.* Since aspects are modular crosscutting units, 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 ("property-based aspects" in AspectJ), make 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.

2.2   Terminology

In discussing aspects, we will use the terminology of AspectJ as it is the best known variant. AspectJ is an extension of Java enabling the specification of aspects and their crosscutting with classes. AspectJ expresses crosscutting through *join points* and *pointcuts.* Joint points are well-defined points in the execution of a program. Pointcuts are distinguished selections of join points that meet some specified criteria. In addition to pointcuts, aspects allow specifying *advice*: method-like constructs defining complementary behavior at join points. To become effective, a given advice has to be attached to a specific pointcut. Depending on the declaration, advice bodies are executed *before* or *after* a specified join point, or they can surround (*around*) that join point. Besides pointcuts and advice, aspects can list ordinary Java member declarations such as attributes, methods, and constructors.

3   Design by Contract

In this section we present the concepts of *Design by Contract* (DbC) [9] and introduce a running example.

3.1   Overview

DbC is a methodology to guide analysis, design, documentation, testing and other tasks of software development.

The application of DbC to software design requires the developer to build software systems based on precisely defined contracts between cooperating components. Contracts, similar to real-life contracts, define the relationship between two cooperating components — the client and the supplier — by expressing each party's expectations and guarantees. A general characteristic of contracts is that they turn the obligation for one of the parties into a benefit of the other. Contracts are expressed through *preconditions*, *postconditions*, and *class invariants*.

– *Preconditions* are conditions that the client must fulfill for the supplier to carry out its task properly. Preconditions are an obligation for the client and a benefit for the supplier.
– *Postconditions* are properties satisfied at the end of the task's execution, assuming the preconditions were met at the beginning. Postconditions are an obligation for the supplier and a benefit for the client.
– *Class invariants* capture the deeper semantic properties and integrity constraints of a class and its instances. Class invariants must be satisfied by every instance of a class whenever the instance is externally accessible: after creation; before and after any call to an exported routine (function or procedure) of the class.

Applying DbC to software analysis and design helps build correct software. By providing preconditions, postconditions and class invariants, programmers explicitly specify the assumptions on which they rely to ensure the correctness of every software element.

Besides providing analysis and design guidance, DbC improves the documentation of software systems. Preconditions describe to client programmers which conditions they must achieve before calling a routine. Postconditions describe what the routine will then do for them. Class invariants make it possible to reason abstractly and statically on the objects of a system's execution. The "contract view" of a class, retaining these elements but discarding implementations, provides a description of software components at just the right level of abstraction; it can be produced by tools of the environment, such as EiffelStudio, and serves as the fundamental form of documentation for DbC-based software, making it possible to use a component on the sole basis of its contract view. In the DbC view all this assumes — as noted above — that the contracts are an integral part of the software text.

DbC also plays a major role in the debugging and testing of software systems. As preconditions, postconditions and invariants are correctness conditions governing the correct functioning of software components and the relationship between components, a contract violation at run time always signals a bug: client bug for a precondition violation, supplier bug for a postcondition or invariant violation. A DbC development environment must provide a compilation option for enabling and disabling run-time contract monitoring. This provides crucial help during the testing and debugging process, making these activities more focused than in other approaches: only with explicit contracts can one explicitly compare the relationship between desired and actual behavior.

Among other advertised benefits of contracts is their support for the maintenance activity: when changing a class, the maintainer should be guided by the existing contracts, class invariants in particular, and generally maintain them even if the implementation changes.

Contracts also play a significant role, as detailed in section 5.4, in controlling the inheritance mechanism, including when used for analysis of software.

3.2   An Example

In this section we introduce the running example of this paper. The example is based on one presented in [4] and models points in the Cartesian coordinate system. To illustrate the basic DbC ideas we provide the example first in *Eiffel* [10]. Later, in section 4, we show the corresponding implementation in AspectJ.

Listing 1.1 shows the Eiffel class POINT implementing the example. This version does not contain any contracts yet (see Listing 1.2 for the version including contracts). The class declares the following five features[2]: the attributes x

---

[2] Features are the basic constituents of Eiffel classes and are either attributes or routines.

```
class
    POINT

feature —— Access

    x: DOUBLE
          —— Abscissa value of current point
    y: DOUBLE
          —— Ordinate value of current point

feature —— Element change

    change_point (new_x: DOUBLE; new_y: DOUBLE) is
          —— Overwrite coordinates of current point with provided values.
       do
          x := new_x
          y := new_y
       end

    move_by (delta_x: DOUBLE; delta_y: DOUBLE) is
          —— Move current point by provided values.
       do
          x := x + delta_x
          y := y + delta_y
       end

feature —— Output

    to_string: STRING is
          —— String representation of current point
       do
          Result := "(" + x.out + ", " + y.out + ")"
       end

end
```

Listing 1.1. Cartesian point implementation in Eiffel without contracts

and y, the procedures change_point and move_by, and the function to_string. The various feature clauses within the class text allow feature categorization.

All features in POINT are accessible by all clients of the class. Attributes x and y are public; this is the appropriate policy since attribute access in Eiffel is read-only for clients (any modification requires, in accordance with object-oriented principles, the provision of an associated "setter" procedure).

Listing 1.2 shows the complete point implementation with contracts. Due to the straightforwardness of the example there are only a few contracts, in this case only postconditions. In Eiffel, postconditions are introduced by the keyword **ensure** following the routine body. Preconditions are introduced by the keyword **require** and have to precede the routine body, that is they are placed before the **do** keyword. Class invariants are introduced by the keyword **invariant** and have to be mentioned after the last feature in the class text. It is possible, although not required, to *tag* individual assertion clauses for identification: the example uses this possibility, as with the y_correctly_moved tag for the last assertion of the procedure move_by.

The class in listing 1.1 appears satisfactory in some general and fairly vague sense — vague precisely because in the absence of contracts it is impossible to know for sure what it is really supposed to do, and hence to talk about its

```
class
    POINT

feature —— Access

    x: DOUBLE
            —— Abscissa value of current point
    y: DOUBLE
            —— Ordinate value of current point

feature —— Element change

    change_point (new_x: DOUBLE; new_y: DOUBLE) is
            —— Overwrite coordinates of current point with provided values.
        do
            x := new_x
            y := new_y
        ensure
            x_correctly_updated: x = new_x
            y_correctly_update: y = new_y
        end

    move_by (delta_x: DOUBLE; delta_y: DOUBLE) is
            —— Move current point by provided values.
        do
            x := x + delta_x
            y := y + delta_y
        ensure
            x_correctly_moved: x = old x + delta_x
            y_correctly_moved: y = old y + delta_y
        end

feature —— Output

    to_string: STRING is
            —— String representation of current point
        do
            Result := "(" + x.out + ", " + y.out + ")"
        end

end
```

Listing 1.2. Cartesian point implementation in Eiffel with contracts

"correctness" or absence thereof. The new variant of POINT in listing 1.2 keeps the implementation — so that in non-erroneous cases at least the new class will "function" like the previous one — but makes it possible to talk about correctness since it now includes contracts.

The postcondition of procedure move_by, for example, uses old x to denote the value attribute x had on routine entry, that is before the assignment was executed. Such "old expressions", permitted only in postconditions, are essential to express the effect of a routine by stating how the final state differs from the original state.

Listing 1.2 demonstrates the power of having contracts embedded within the class text. Although contracts are clearly distinguishable from the actual routine bodies, they are still part of their declaration. This encourages programmers to update contracts when changing implementations. If they forget, run-time contract monitoring will help spot the mistake quickly.

4 Aspectizing Contracts

In an interview [5] Kiczales cites DbC as an example of a crosscutting concern:

"[...] there are many other concerns that, in a specific system, have cross-cutting structure. Aspects can be used to maintain internal consistency among several methods of a class. They are well suited to enforcing a Design by Contract style of programming."

Other authors [4, 6, 7, 8] share that opinion. The idea has also been patented [6]. Let us pursue this idea and attempt to aspectize the preceding example using AspectJ.

Listing 1.3 displays a Java counterpart of the original, uncontracted Eiffel class POINT of listing 1.1. Except for syntactical differences, the two versions most notably differ in the handling of attributes (fields in Java terminology). Since attribute access in Java is not restricted by default, Java programmers typically declare attribute values to be private and, additionally, provide appropriate "getter" methods.

Listing 1.4 displays the aspectized version of listing 1.3 that emulates the contracted Eiffel version of listing 1.2 within an aspect using AspectJ. For simplicity, we only list the aspectized postcondition of procedure move_by(); the other contract clauses would be amenable to the same treatment.

The aspect PointMoveByContract defines a pointcut, which denotes all calls of moveBy(), and an associated **after** advice for the postcondition enforcement. In case the postcondition is violated, the advice throws an exception. In addition, the aspect declares auxiliary constructs for recording the attribute values before

```java
public class Point {

    private double x;
    private double y;

    public double getX(){
        return x;
    }

    public double getY(){
        return y;
    }

    public void changePoint(double x, double y){
        this.x = x;
        this.y = y;
    }

    public void moveBy(double deltaX, double deltaY){
        this.x += deltaX;
        this.y += deltaY;
    }

    public String toString(){
        return "(" + getX() + ", " + getY() + ")";
    }
}
```

Listing 1.3. Cartesian point implementation in Java

```
public aspect PointMoveByContract {

    private double Point.oldX;
    private double Point.oldY;
    private double Point.newX;
    private double Point.newY;

    public pointcut checkContract(Point p, double dx, double dy):
        call(void Point.moveBy(double, double))
        && args(dx, dy)
        && target(p);

    before(Point p, double dx, double dy): checkContract(p, dx, dy){
        p.oldX = p.getX();
        p.oldY = p.getY();
    }

    after(Point p, double dx, double dy): checkContract(p, dx, dy){
        p.newX = p.getX();
        p.newY = p.getY();
        if ((p.newX != p.oldX + dx) || (p.newY != p.oldY + dy))
          throw new ContractViolationException("p not correctly moved");
    }
}
```

Listing 1.4. Aspect implementing the postcondition of moveBy()

and after the method moveBy() was executed. To this end, the aspect includes the inter-type declarations Point.oldX, Point.oldY, Point.newX, and Point.newY and a **before** advice.

Although other techniques might be available, this seems to be the most direct and effective way to "aspectize" contracts as suggested by the AOP literature.

5   Analysis

To compare the original contracted version (listing 1.2) and its aspect-based emulation, we examine it in the light of some of the benefits of contracts listed in section 3.1 — analysis and design, documentation, testing and debugging — and two other important criteria: reusability and ease of use.

5.1   Analysis and Design Guidance

Both approaches provide design guidance by requiring developers to build software systems based on precisely defined contracts. In AOP, however, one tends to impose aspects on classes once the classes are developed, which clearly contradicts the DbC methodology.

5.2   Documentation Aid

As Eiffel includes preconditions, postconditions, and invariants directly in the class text, contracts become part of the class specification and thus increase system documentation. In AspectJ, contracts are separated from the class they describe and thus force the programmer to switch back and forth between classes

and aspects. Appropriate tool support, such as the AspectJ Development Tools (AJDT) project for the Eclipse development environment, alleviates the problem by highlighting the places where advice are injected.

5.3   Testing Assistance

Eiffel relies on run-time contract monitoring, which can be enabled or disabled by setting a respective compilation option. AspectJ too, facilitates "pluggable" run-time contract monitoring since aspectized contracts can be added or removed from classes. Once an aspectized contract is removed, however, its documentation is lost.

5.4   Contract Reuse

The application of wildcards in pointcuts makes aspects applicable to many classes. In this way, AspectJ facilitates contract reuse. Eiffel too, allows contract reuse, but restricts it to classes related by inheritance (see section 6.4).

5.5   Ease of Use

Eiffel promotes ease of contract application. Programmers declare contracts directly within the class text, at the place where they want them to apply. Moreover, Eiffel provides several constructs, such as the old notation (see section 3.2), to facilitate the expressing of assertions. In AspectJ, programmers specify the code places, where to inject the contracts, indirectly by means of pointcuts.

6   Discussion

In this section, we attempt to generalize from the example and assess whether aspects are suited for emulating DbC.

6.1   Support of DbC Methodology

The analysis in section 5 implies that aspect-based DbC implementations do not support the DbC methodology to the same extent as implementations in contract-enabled languages. An aspect-based emulation ignores the documentation mechanisms made possible by DbC. Since contracts are separated from classes, the risk of introducing inconsistencies between classes and contracts is increased; as programmers become aware of contracts only when using special tools for browsing the spread application structure, they are more likely to forget adapting the contracts when changing the classes. This issue can get worse after system deployment, once the aspectized contracts are removed. How will a client, of a library component for example, know under which conditions it may call a service?

Such removal of contracts also raises questions of maintainability. One of the benefits of contracts mentioned in section 3.1 is to guide the evolution of classes; but this assumes that the contracts are in the software.

Aspectized contracts promote a different focus on software design than DbC. Whereas AOP aims at separating concerns, DbC fosters the explicit specification of inter-module cooperation, in the modules themselves.

6.2   Are Contracts Crosscutting?

Contracts are *recurrent* in DbC-based code. Are they also *crosscutting*?

We can assess this conjecture by examining its consequences. If contracts were crosscutting, an aspectized contract implementation would yield the benefits presented in section 2.1. For example, the improved *modularity* requires aspectized contracts to be modular. According to Parnas [14] modularity is, besides the existence of explicit interfaces, defined by the possibility to create a module with little knowledge of the code in other modules and the expectation that drastic changes within one module will not require adaptations in other modules. Contracts do not meet that definition: Since they semantically depend on their classes, they can only be created with detailed knowledge of the associated classes, and are likely to be affected by modifications of these classes. Aspectization is unlikely to change that situation.

Similar reasoning applies to other benefits listed: *evolution* and *stability*. Separating contracts from classes does not isolate the changes.

The remaining benefits mentioned in section 2.1 — *explicitness*, *reusability*, *pluggability* — are achieved, at least partially, by an aspectized contract implementation. They would, however, also exist without aspectization. Implementing contracts in Eiffel (see listing 1.2) makes contracts explicit, offers pluggability of run-time contract monitoring, and allows contract reuse through genericity and multiple inheritance (see also section 6.4 below).

The preceding considerations suggest that although contracts appear repeatedly within the class text — before and after each routine and at the end of the class — they are not crosscutting by nature.

6.3   Aspect Interactions

Hannemann et al. [15] and Bergmans [16] have pointed out the existence of aspect interactions in AOP. Such interactions can result in conflicts between classes and aspects or between aspects and aspects. Aspects intended to emulate contracts seem to create many such interactions.

To illustrate that problem, we extend our running example and assume that we want to scale points transparently to the clients of class Point. This is typical of the kind of incremental, seamless addition that aspects are intended to permit. Listing 1.5 presents an aspect achieving this extension for scaling points according to a specified scale factor. Whenever we attempt to change the coordinates of a point, the **around** advice of aspect ScalePoint will multiply the new

```
public aspect ScalePoint {

    private static double scaleFactor = 10;

    public pointcut change(Point p, double x, double y):
        target(p) &&
        args (x, y) &&
        (execution(void Point.changePoint(double, double)) ||
                execution(void Point.moveBy(double, double)));

    void around(Point p, double x, double y): change(p, x, y){
        proceed(p, scaleFactor * x, scaleFactor * y);
    }
}
```

Listing 1.5. Aspect allowing to scale points by a specified scale factor

coordinates we provide by ₛcₐₗₑFₐctₒᵣ. Such an aspect could be useful for displaying points when the display device requires a special formatting.

Adding such an aspect breaks the contract of class ₚₒᵢₙt. Both the new aspect and the aspectized contract advise method ₘₒᵥₑBy. The problem is that they work on the same method in a nested fashion without being aware of each other. As soon as one aspect changes the state of the object on which the routine operates, or changes the value of an argument of the routine, it compromises any assumptions by the other aspect on object state or argument values.

The interleaved advice execution sequence in the example is as follows:

1. Before advice of aspect ₚₒᵢₙtMₒᵥₑByCₒₙtᵣₐct.
2. Around advice of aspect ₛcₐₗₑPₒᵢₙt.
3. After advice of aspect ₚₒᵢₙtMₒᵥₑByCₒₙtᵣₐct.

Since the around advice of aspect ₛcₐₗₑPₒᵢₙt changes the point coordinates invisibly to the aspectized contract, the postcondition ceases to be ensured. With contract monitoring on, the after advice will raise a CₒₙtᵣₐctVᵢₒₗₐtᵢₒₙExcₑₚtᵢₒₙ. The aspect ₛcₐₗₑPₒᵢₙt interferes with the contract.

The example suggests that no module in a system — class or aspect — can be oblivious of the presence of contracts.

6.4   Contracts and Inheritance

A key property of DbC is its connection with the inheritance mechanism and, as a consequence, software reuse. When a class inherits from one or more others, the following rules apply to its contracts [9]:

– *Parent's invariant rule:* The invariants of all the parents of a class apply to the class itself.
– *Assertion redeclaration rule:* A routine redeclaration may only replace the original precondition by one equal or weaker, and the original postcondition by one equal or stronger. This applies both to *redefinition* (overriding an inherited implementation) and *effecting* (providing a first implementation of a feature inherited as *deferred*, that is to say, without an implementation);

the second application is particularly useful in the transition from analysis (which uses deferred classes) to design and implementation.

Ideally, the development environment should enforce these rules. With aspectized contracts, however, there seems to be no clear way of achieving this. It is left to the programmer to make sure that the pointcuts for injecting invariants also apply to descendant classes and that the refinement of preconditions and postconditions in descendant classes follows the assertion redeclaration rule. Intended revisions on the AspectJ join point model aiming at increasing the expressiveness of pointcut definitions [17] might abate the problem in future. For the moment, however, programmers must live with the pure syntactical mechanisms of the AspectJ join point model, and take care of consistent contract refinement themselves.

7   Conclusion

Our investigation of whether aspects are suited for implementing Design by Contract suggests that such an emulation fails to provide some of the principal benefits of Design by Contract, in particular documentation and support for inheritance-based refinement. We have further observed that contracts are not crosscutting by nature, and that aspectizing them risks creating interactions with other aspects.

These conclusions are of course dependent on the context of our study: it may be — although we have no evidence of either possibility — that using another AOP environment than AspectJ, the current flagship and reference for AOP, would yield better results; and that we used the wrong techniques for aspectizing contracts, missing more effective solutions.

Based on the current state of our aspectizing efforts, however, the conclusion seems clear: the widely repeated AOP claim that aspects can emulate contracts does not appear to stand.

References

1. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Aksit, M., Matsuoka, S., eds.: ECOOP '97 - Object-Oriented Programming: 11th European Conference. Volume 1241 of Lecture Notes in Computer Science., Springer-Verlag GmbH (1997) 220–242
2. Lieberherr, K.J., Lorenz, D.H., Mezini, M.: Building modular object-oriented systems with reusable collaborations (tutorial session). In: ICSE, ACM Press (2000) 821
3. Lopes, C.V., Kiczales, G.: Improving design and source code modularity using AspectJ (tutorial session). In: ICSE, IEEE-CS : Computer Society and SIGSOFT: ACM Special Interest Group on Software Engineering and Irish Comp Soc : Irish Computer Society, ACM Press (2000) 825
4. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: Getting started with AspectJ. Commun. ACM **44** (2001) 59–65

5. TheServerSide.COM: Interview with Gregor Kiczales, topic: Aspect-oriented programming (AOP). http://www.theserverside.com/talks/videos/GregorKiczalesText/interview.tss (2003)
6. Lopes, C.V., Lippert, M., Hilsdale, E.A.: Design by contract with aspect-oriented programming. U.S. Patent No. 6,442,750 (2002)
7. Diotalevi, F.: Contract enforcement with AOP.
   http://www-128.ibm.com/developerworks/library/j-ceaop/ (2004)
8. Skotiniotis, T., Lorenz, D.H.: Cona: aspects for contracts and contracts for aspects. In: OOPSLA Companion, ACM Press (2004) 196–197
9. Meyer, B.: Object-Oriented Software Construction. Second edn. Prentice Hall Professional Technical Reference (1997)
10. Meyer, B.: Eiffel: The Language. Prentice Hall Professional Technical Reference (1991)
11. Kiczales, G.: AspectJ: Aspect-oriented programming in Java. In Aksit, M., Mezini, M., Unland, R., eds.: NetObjectDays. Volume 2591 of Lecture Notes in Computer Science., Springer-Verlag GmbH (2002) 1
12. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In Knudsen, J.L., ed.: ECOOP. Volume 2072 of Lecture Notes in Computer Science., Springer-Verlag GmbH (2001) 327–353
13. Lopes, C.V., Kiczales, G.: Recent developments in aspect. In Demeyer, S., Bosch, J., eds.: ECOOP Workshops. Volume 1543 of Lecture Notes in Computer Science., Springer-Verlag GmbH (1998) 398–401
14. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15** (1972) 1053–1058
15. Hannemann, J., Chitchyan, R., Rashid, A.: Analysis of aspect-oriented software. In Buschmann, F., Buchmann, A.P., Cilia, M.A., eds.: ECOOP Workshops. Volume 3013 of Lecture Notes in Computer Science., Springer-Verlag GmbH (2003) 154–164
16. Bergmans, L.: Towards detection of semantic conflicts between crosscutting concerns. In Hannemann, J., Chitchyan, R., Rashid, A., eds.: Workshop on Analysis of Aspect-Oriented Software. ECOOP 2003 (2003)
17. Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In: ICSE, ACM Press (2005) 49–58