AOP vs Software Evolution:
a Score in Favor of the Blueprint.

Walter Cazzola[1] and Sonia Pini[2]

[1] Department of Informatics and Communication,
Università degli Studi di Milano, Italy
`cazzola@dico.unimi.it`
[2] Department of Informatics and Computer Science
Università degli Studi di Genova, Italy
`pini@disi.unige.it`

Abstract. All software systems are subject to evolution, independently by the developing technique. Aspect oriented software in addition to separate the different concerns during the software development, must be "not *fragile*" against software evolution. Otherwise, the benefit of disentangling the code will be burred by the extra complication in maintaining the code.
To obtain this goal, the aspect-oriented languages/tools must evolve, they have to be less coupled to the base program. In the last years, a few attempts have been proposed, the Blueprint is our proposal based on behavioral patterns.
In this paper we test the robustness of the Blueprint aspect-oriented language against software evolution. Keywords: AOP, Software Evolution, Fragile Pointcut Problem.

1   Introduction

All software systems are subject to evolution, they evolve over time as new requirements and functionality emerge, or adaption and extensions are to be made. Studies pointed out that up to 80% [13] of the system lifetime will be spent on maintenance and evolution activities. A program that is useful in a real-world environment necessarily must change or become progressively less useful in that environment [12].

Aspect-oriented programming has been designed with the intention of providing a better separation of concerns by modularizing concerns that would otherwise be tangled and scattered across the other concerns. This would made the software more maintainable, evolvable and understandable. Paradoxically, the major aspect-oriented techniques instead of improving software maintainability seem to restrict the evolvability of that software, as highlighted in [21]. This problem is due to the so called "fragile pointcut problem" [11].

Pointcuts are deemed fragile when seemingly innocent changes to the base program, such as renaming or relocating a method, break a pointcut such that it no longer captures the join points it is intended to capture. When code is added to a program and introduces new join points in the program, pointcuts are similarly considered fragile in the case some of these new join points should be captured by the pointcut but it fails to do so.

It implies that all pointcuts of each aspect need to be checked and possibly revised whenever the base program evolves, since they often break when the base program is re-factored. All pointcuts referring to the base program need to be examined both after an evolution and after a re-factoring, because they capture a set of join points based on some structural and syntactical properties, any change to the structure or syntax of the base program can alter the set of join points that is captured by the pointcuts. This problem exists both if a programmer uses wildcards and not.

In practice, the pointcut fragility derives form the *dependency* on the program syntax and the *coupling* between aspects and program [1, 7, 8, 20]. The fragile pointcut problem is a serious inhibitor to evolution of aspect-oriented programs. The critical element of the past generation of AO tool is the necessity to specify program elements names and the impossibility to select elements without using naming convention or regular expressions. In short, they have a *linguistic approach*, so aspect writers need to be completely aware of the base-code details and evolution, so each aspect become strictly bound to the application on who has been designed. To obtain a less fragile approach, it is necessary a new generation of aspect-oriented languages/tools, less coupled to the base program.

In the last few years, the main goal of the new generation of AO approaches is to get a more semantic join point selection mechanism to avoid the fragile pointcut problem. Some approaches are: the pointcut delta analysis [11, 20], the approach of Kellens et al. described in [8], the join point model proposed by Mohd Ali and Rashid in [14], the functional query language proposed by Eichberg et al. in [4], the graphical approach to model pointcuts described by Stein et al. in [19] and so on. In [3, 16] we have defined a new (visual) model-based join point selection mechanism. We tackle the fragile pointcut problem by eliminating the intimate dependency of pointcut definitions on the base program and by using a high level description of the program behavior during the join point selection.

In this paper we want to prove the robustness of the Blueprint against the evolution. The rest of the paper is organized as follows: in section 2 we shortly overview the Blueprint approach and elements, in section 3 we introduce our test case for the evolution, finally, in section 4 and in section 5 we face some related works and draw out our conclusions.

2   The Blueprint Language

The Blueprint framework is based on our previous work [2] and it is completely detailed in [16]. Moreover, a working prototype of the framework has been developed in Java.

The main goal of the Blueprint language is to overcome many problems of the past generation of AO language [1, 9, 11], such as, the granularity of the join point, the fragile pointcut problem, the semantic selection and so on.

The Blueprint is based on the idea that the description of the application behavior cannot be strictly coupled to the application syntactic details. It permits a *loose approach* to the description of the application behavior. This means that the aspect programmer can use different levels of detail during the description of a single join point blueprint

by using any possible combinations of *loose* and *tight elements*. This approach permits to describe a well identified behavior tightly coupled to the application code by specifying the names of the involved elements, and a less known behavior by using *meta-information* to abstract from the real application code.

The Blueprint is a novel aspect-oriented framework, its join point selection mechanism allows the selection of the join points *abstracting* from implementation details, name conventions and any other source code dependency. In particular the aspect programmer can select the interested join points by describing their supposed location in the application through UML-like descriptions (basically, activity diagrams) representing computational patterns on the application behavior; these descriptions are called blueprints. The blueprints are just patterns on the application behavior, i.e., they are not derived from the system design information but express properties on them. In other words, we adopt a sort of enriched UML diagrams to describe the application control flows or computational properties and to locate the join points inside these contexts.

The Blueprint uses a static quantification, i.e., it allows quantification over the abstract syntax tree of the program, hereby queries such as "print the value of a variable used in a loop test condition and modified in the loop body" are possible. This kind of quantification requires to access the source code of the application, because we need to obtain a parsed version of the underlying program, to run the transformation rules realizing the quantified aspects over that abstract syntax tree. The Blueprint language can be used on the bytecode as well since it can be univocally decompiled (modulo semantic equivalence) by apposite tools, e.g., by Jode.

In our approach, we do not need to use position qualifiers such as `before` and `after` advices to indicate where to insert the concern inside the base code, since we describe the context we are looking for, we can either locate the join points exactly where we want to insert the new code or, to highlight the portion of behavior we want to replace.

The Blueprint framework recalls the AspectJ terminology but some terms are used with a slightly different meaning. The Blueprint *join points* are *hooks where code may be added* rather than *well defined points in the execution of a program* as in AspectJ. In other words, the AspectJ join points are based on the idea that "when something happens, then something gets executed[3]." In this view a join point consists of things like method and constructor calls, method and constructor executions, object instantiations, field references and so on. While the Blueprint approach is that "join points can occur in any part of the code", this view permits of changing a single line of code. We use a *statement-level granularity* for the join point model whereas AspectJ uses an *operational level granularity* for the join point model. In particular we consider two different kinds of join points: the *local join points* that represent points in the application behavior where to insert the code of the concern, and *region join points* that represent portion of the application behavior that must be replaced by the code of the concern. The pointcuts are obtained as composition/enumeration of the join points selected by the join point blueprints rather than as the logical composition of queries on the application code. While *introduction*s and *advice*s keep their usual meaning.

To complete the picture of the situation, we have introduced some new concepts: *join point blueprint* and, *blueprint space*. The former is *a template (a blueprint) on*
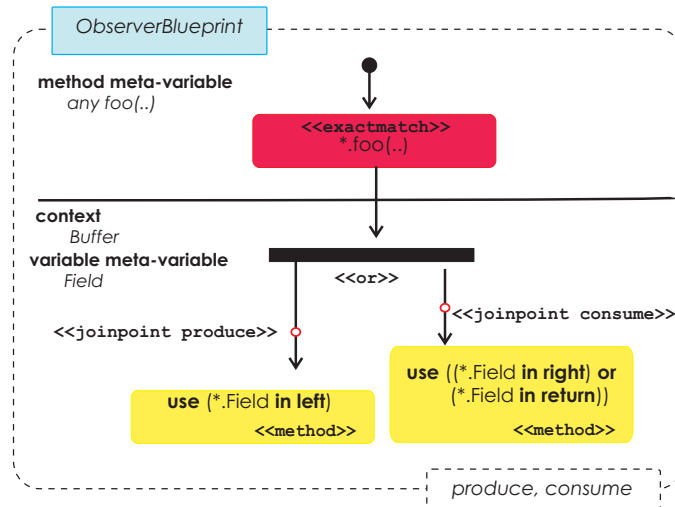
---

[3] `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`

Fig. 1. Sample Join Point Blueprint.

the application behavior identifying the join points in their context; these blueprints describe where the local and region join points should be located in the application behavior. The blueprint does not completely describe the computational flow but only the portions relevant for selecting the join points. The latter is the set of all join points blueprints defined on the same application.

The key element of our approach are the *join point blueprints*, they graphically depicts where a join point (both local and region) should be in the application behavior. They look like an activity diagrams.

The diagram contextualizes the join point location by describing some crucial *events* that should occur close to the join point, these events will be used to recognize the join point. The frame gives some ancillary information, such as the blueprint name (at the top left corner), the join points name exposed by the blueprint (at the bottom right corner) and some meta-info used by the weaver to parametrize the context and to get values from the join point. The listed join points are the only exposed to the pointcut specification. The join point position is denoted by the «joinpoint name» stereotype (or by the couple «startjoinpoint name» and «endjoinpoint name» for the region join points).

To get a more expressive language and less coupled to the code syntax and structure, and by assuming that the aspect programmer does not necessarily know the implementation details of the code, we introduced the *meta-information* section inside the blueprint diagram. The meta-information is the textual portion of the blueprint, that allows the programmer of describing the context at the desired implementation detail, i.e., either by using real variable, method and field names or less precise information. For example, if the programmer does know the method name, but he knows its parameter types, he can use this information, in the meta-information section, by declaring a new

method meta-variable with a fantasy name,and indicating the right number and type of its parameters, and finally by using this new meta-variable in the blueprint to describe the sought behavior. The meta-information elements will be unified to variable names used into the application during the *weaving phase*.

Figure 1 shows a very simple join point blueprint that absolutely do not recall the whole expressivity allowed by the formalism. For a detailed and exhaustive description, please, refer to [16] chapter 4.

3 A Test-Bed for the Blueprint Robustness

So far, we have used the Blueprint to locate simple join points in toy-applications, like showed in [3,16], with few lines of code. Now, it is fundamental to test the robustness of the Blueprint language against the evolution by using a real application with thousand lines of code and a long time life cycle with many adaptation steps. To this goal, we adopted the Health Watcher (HW) system[4] developed at UPE and introduced by Soares et al. in [18].

HW is a typical web-based application that manages health-related information. It includes a variety of crosscutting concerns, such as concurrency, distribution, persistence, and so on. The same application has been previously used as test-bed by the Lancaster University (in [6]), that has created and compared one object-oriented (by using Java) and two aspect-oriented (by using AspectJ and CaesarJ) implementations of HW. Moreover, they introduced nine steps of evolution to the initial application.

3.1 HW Evolution

We consider the HW evolution from version 8 to version 9, which adds new functionalities to the application. Version 9 adds the following functionality: insertion of new health unit, insertion of new medical specialization, insertion of new symptoms, searching for symptoms, updating of a symptom, searching for specialization by code, updating of medical specialization, and insertion of new disease types.

These new functionalities involve the creation of new records and repositories for *diseases* and *symptoms*. Potentially, they can introduce changes to the public interface and interfere with the correct working of the existing pointcuts.

Most of the AOP approaches use a join point model similar to that of AspectJ [10]. The AspectJ pointcut language offers a set of *primitive pointcut designators*, such as `call`, `get` and `set` specifying a method call and the access to an attribute. All the pointcut designators expect, as an argument, a string specifying a pattern for matching method or field signature. These string patterns introduce a real dependency of the syntax of the base code. Intuitively, since pointcuts capture a set of join points based on some *structural* or *syntactical* property, any change to the structure or syntax of the base program could also change the applicability of the pointcuts and the set of captured join points.

---

[4] The complete source code developed is available at http://www.comp.lancs.ac.uk/ greenwop/-tao

Aspect developer implicitly imposes some *design rules* that the base program developer has to follow when evolves his program to be compliant with the existing aspects and avoid of selecting more or less join points than expected. In this case, problems with evolution depend also of the need of guessing these, often silent, conventions. These rules derive from the fact that pointcuts often express *semantic properties* about the base program in terms of its *structural properties*.

First to present our approach, we present the problems encountered, also in this case, by the AspectJ aspects. In particular, we consider the aspect used for the HW synchronization of concurrent insertion and showed in Listing 1.1.

It is fairly evident that the pointcut definition takes in consideration only the method name of a particular class and not the behavior or the semantic to locate the interested join points. In this case, the aspect programmer has written a correct pointcut, and the corresponding aspect works as intended. When the code is changed (i.e., in version 9) by adding new *persistent entity*, i.e., `DiseaseRecord` and `SymptomRecord` despite the behavior added by these new entities is the same of the old entity, the `synchronizationPoints` pointcut (see Listing 1.2) has been changed in order to consider also the new methods.

Listing 1.1. The AspectJ pointcut in version 8

```
public pointcut synchronizationPoints(Employee employee) :
  execution(* EmployeeRecord.insert(Employee))
    && args(employee);
```

Note that this is only a possible way to write the pointcut, but in general the problems are the same. For example, in Listing 1.1, we can insert a wildcard in place of the class name (`EmployeeRecord`), in this case the pointcut is not broken by the evolution, but if the programmer decides to change the method name, e.g., from `insert` to `store` the pointcut does not work right, and (s)he must adapt the pointcut definition to locate all the right point in the application.

Listing 1.2. The AspectJ pointcut in version 9

```
public pointcut synchronizationPoints(Object o) :
  (execution(* EmployeeRecord.insert(Employee)) ||
  execution(* DiseaseRecord.insert(DiseaseType)) ||
  execution(* SymptomRecord.insert(Symptom)) )&& args(o);
```

Since, the problem of the evolution in aspect-oriented programs is mainly that the set of join points captured by a pointcut may change when changes are made to the base program, even though the pointcut definition itself remains unaltered. Then, to avoid this problem we need a *low coupling* of the pointcut definition with the source code. The aim of the Blueprint approach is just to overcome the AOP problem about software evolution, by allowing the selection of the join points abstracting from *implementation details*, *name conventions* and any other *source code dependency*.

In Figure 2 is showed the join point blueprint used to locate the methods that need synchronization, it describes a *relevant portion of a method behavior*. In particular, since all application methods that store records into repositories are composed by a check to
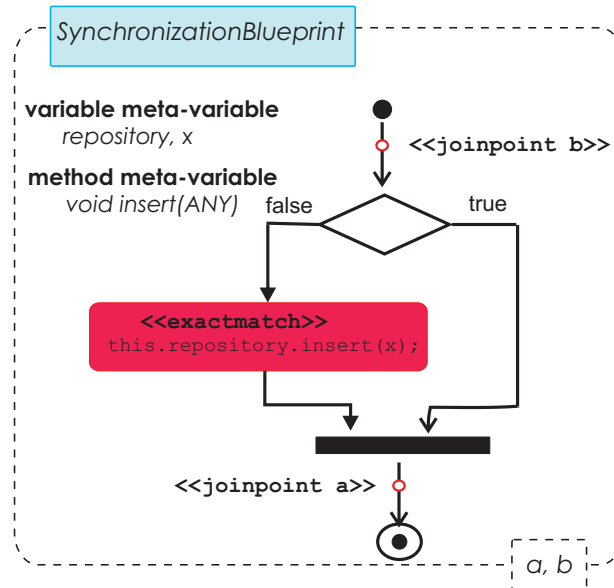
Fig. 2. Join Point Blueprint for Insertion Synchronization.

control if the record is already inserted or not inside the repository, we can search an `if` statement containing, in the false branch, the code to insert the record.

The «`joinpoint b`» locate the join point at the beginning of the method that contains the statement that match the relevant portion of behavior, while the «`joinpoint a`» locate the join point at the end of the method that contains the matched statements of the diagram. The diamond indicate that we are looking for a conditional statement, where the condition is not relevant for the context definition, like so, it is not relevant what is contained in the `true` branch.

The `repository` variable meta-variable used in the action (i.e., the red rounded rectangle) during the weaving process is unified to a class field. The `insert` method meta-variable represents a method that does not return nothing and that has only one parameter of any type Note that the name of the method meta-variable is completely independent from the name of the searched method, i.e., changing the name `insert` in `abcdef` the located behavior and unified variables do not change.

Since the new entities have almost the same behavior of the old one, like showed in Listing 1.3, the behavior described in the blueprint locate all the methods that need a synchronization point.

Our selection mechanism matches the `insert()` method of the `EmployeeRecord` class presents in version 8, and the `insert()` methods present in the `DiseaseRe-cord` and, `SymptomRecord` classes, added since version 9, without change anything. The `repository` variable meta-variable is respectively unified to `employeeReposi-tory`, `diseaseRep` and `rep` application's field. The `void insert(ANY)` method meta-

Listing 1.3. The Application Implementation

```
// insert method of EmployeeRecord class
public void insert(Employee employee) // throws clause  {
  if (employeeRepository.exists(employee.getLogin())) {
    throw // new exception  ;
  } else {
    employeeRepository.insert(employee);
  }

// insert method of DiseaseRecord class
public void insert(DiseaseType td) // throws clause  {
  if (diseaseRep.exists(td.getCode())) {
    throw // new exception  ;
  } else {
    this.diseaseRep.insert(td);
  }

// insert method of SymptomRecord class
public void insert(Symptom symptom) // throws clause  {
  if (rep.exists(symptom.getCode())) {
    throw // new exception  );
  } else {
    rep.insert(symptom);
  }
}
```

variable is respectively unified to the `insert` method of the `EmployeeRepositor-yArray`, `DiseaseTypeRepositoryArray`, and `SymptomRepositoryArray` class.

4   Related Works

The Blueprint framework is not the first attempt of dealing with the limitations of the current join point selection mechanisms. In the next of the section we report some of the most significant attempts, without pretending to be exhaustive.

In [8], Kellens et al. tackle the *fragile pointcut problem* by replacing the intimate dependency of pointcut definitions on the base program by a more stable dependency on a *conceptual model* of the program. This conceptual model provides an abstraction over the structure of the source code and classifies base program entities according to the concepts that they implement. The strength of the approach is on the definition of the conceptual model of the base program. The classification of source-code entities in the conceptual model is constructed using annotations in the source code and, defining extra design constraints that need to be respected by source-code entities, for the model to be consistent. This approach requires developers to describe a conceptual model of their program and its mapping to the program code, in this way, it breaks the *obliviousness* [5] property. Moreover, it is coupled with the structure of the base program,

but not coupled with its implementation, and only the program entities can be used to define pointcut, since they use the same join point of the AspectJ join point model. Finally, they still need a mechanism for automatically verifying the correctness of the classifications defined by the conceptual model.

In [4], Eichberg, et al. present the usage of functional query language for the specification of pointcuts. In their approach a pointcut is a set of nodes in a tree representation of the program's modular structure, and this set is selected by query on node attribute written in a query language. They created an XML-to-class file assembler/disassembler that can be used to create an XML representation of a class file and convert an XML file back into a class file on the basis of their bytecode framework. The query language is used on top of this XML representation of the program structure. Their join point model defines more join point of the AspectJ one, because bytecode structure permits to identify more point, e.g., the storing of a value in a local variable. Their query language is general enough to express a wide range of very different pointcut models.

In [17], Sakurai and Masuhara propose a new aspect-oriented programming language that uses unit test cases as interface of crosscutting concerns. A test-based pointcut matches join points in the execution of a target program that (potentially) have the same execution history as one of the unit test cases specified by the pointcut. This approach replace the fragile pointcut problem with the maintenance of unit test cases whose cost should anyhow be paid with practical software development.

In [15], Nagy et al. propose a new approach to AOP by referring to program unit through their design intentions to answer to the need of expressing semantic pointcuts. Design intention is represented by annotated design information, which describes for example the behavior of a program element or its intended meaning. Their approach instead of referring directly to the program, provide a new language abstraction to specify pointcuts based on some design information. Design information are inserted inside the base program using annotations and they are associated manually, derived on the presence of other design information and, through superimposition. The key benefit of this approach is that it reduces direct dependencies between the crosscutting concerns and the program source. Unfortunately, this approach breaks the *obliviousness* [5] property. This property is broken because certain design information has to be specified by the software engineer, and moreover the software engineer must use a consistent and coherent set of design information for each sub-domain of an application.

5 Conclusions

Current aspect-oriented approaches suffer from well known fragile pointcut problem. A common attempt to give a solution consists of creating a more semantic mechanism for the join points selection. This paper shortly describe the Blueprint framework, a novel approach to join points identification that permits to decouple aspects definition and base-code syntax and structure. Moreover, this paper presents a test-bed in order to evidence the robustness of the Blueprint pointcut against the software evolution.

6 Acknowledgments

References

1. Walter Cazzola, Jean-Marc Jézéquel, and Awais Rashid. Semantic Join Point Models: Motivations, Notions and Requirements. In *Proceedings of the Software Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT'06)*, Bonn, Germany, on 21st March 2006.

2. Walter Cazzola and Sonia Pini. Join Point Patterns: a High-Level Join Point Selection Mechanism. In Thomas Khüne, editor, *MoDELS'06 Satellite Events Proceedings*, Lecture Notes in Computer Science 4364, pages 17–26, Genova, Italy, on 1st of October 2006. Springer. Best Paper Awards at the 9th Aspect-Oriented Modeling Workshop.

3. Walter Cazzola, Sonia Pini, and Massimo Ancona. Design-Based Pointcuts Robustness Against Software Evolution. In Walter Cazzola, Shigeru Chiba, Yvonne Coady, and Gunter Saake, editors, *Proceedings of the 3rd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'06)*, in 20th European Conference on Object-Oriented Programming (ECOOP'06), pages 35–45, Nantes, France, on 2nd of July 2006.

4. Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as Functional Queries. In *Proceedings of the 2nd ASIAN Symposium on Programming Languages and Systems (APLAS 2004)*, LNCS, Taipei, Taiwan, November 2004. Springer.

5. Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Proceedings of OOPSLA 2000 Workshop on Advanced Separation of Concerns*, Minneapolis, USA, October 2000.

6. Phil Greenwood, Alessandro F. Garcia, Thiago Bartolomei, Sergio Soares, Paulo Borba, and Awais Rashid. On the Design of an End-to-End AOSD Testbed for Software Stability. In *Proceedings of the 1st International Workshop on Assessment of Aspect-Oriented Technologies (ASAT'07)*, Vancouver, Canada, March 2007.

7. Kris Gybels and Johan Brichau. Arranging Language Features for More Robust Pattern-Based Crosscuts. In *Proceedings of the 2nd Int'l Conf. on Aspect-Oriented Software Development (AOSD'03)*, pages 60–69, Boston, Massachusetts, April 2003.

8. Andy Kellens, Kris Gybels, Johan Brichau, and Kim Mens. A Model-driven Pointcut Language for More Robust Pointcuts. In *Proceedings of Software engineering Properties of Languages for Aspect Technologies (SPLAT'06)*, Bonn, Germany, March 2006.

9. Gregor Kiczales. The Fun Has Just Begun. Keynote AOSD 2003, Boston, March 2003.

10. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeff Palm, and Bill Griswold. An Overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, LNCS 2072, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

11. Christian Koppen and Maximilian Störzer. PCDiff: Attacking the Fragile Pointcut Problem. In *Proceedings of the European Interactive Workshop on Aspects in Software (EIWAS'04)*, Berlin, Germany, September 2004.

12. Meir M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9):1060–1076, September 1980. Special Issue on Software Engineering.

13. Meir M. Lehman, Juan Fernandez-Ramil, and Goel Kahen. A Paradigm for the Behavioural Modelling of Software Processes using System Dynamics. Technical Report 2001/8, Imperial College, Department of Computing, London, United Kingdom, September 2001.

14. Noorazean Mohd Ali and Awais Rashid. A State-based Join Point Model for AOP. In *Proceedings of the 1st ECOOP Workshop on Views, Aspects and Role (VAR'05)*, in 19th European Conference on Object-Oriented Programming (ECOOP'05), Glasgow, Scotland, July 2005.

15. István Nagy, Lodewijk Bergmans, Wilke Havinga, and Mehmet Akşit. Utilizing Design Information in Aspect-Oriented Programming. In Robert Hirschfeld, Ryszard Kowalczyk, Andreas Polze, and Mathias Weske, editors, *Proceedings of 4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays)*, LNI 61, pages 39–60, Erfurt, Germany, September 2005.

16. Sonia Pini. *Blueprint: A High-Level Pattern Based AOP Language*. PhD Thesis, Department of Informatics and Computer Science, Università di Genova, Genoa, Italy, June 2007.

17. Kouhei Sakurai and Hidehiko Masuhara. Test-based Pointcuts: A Robust Pointcut Mechanism Based on Unit Test Cases for Software Evolution. In *Proceedings of Linking Aspect Technology and Evolution revisited (LATE'07)*, Vancouver, British Columbia, Canada, March 2007.

18. Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing Distribution and Persistence Aspects with AspectJ. In Mamdouh Ibrahim and Satoshi Matsuoka, editors, *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 174–190, Seattle, Washington, USA, November 2002. ACM Press.

19. Dominik Stein, Stefan Hanenberg, and Rainer Unland. Modeling Pointcuts. In *Proceedings of the AOSD Workshop on Aspect-Oriented Requirements Engineering and Architecture Design*, Lancaster, UK, March 2004.

20. Maximilian Störzer and Jürgen Graf. Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 653–656, Budapest, Hungary, September 2005. IEEE Computer Society.

21. Tom Tourwé, Kris Gybels, and Johan Brichau. On the Existence of the AOSD-Evolution Paradox. In *Proceedings of the Workshop on Software-engineering Properties of Languages for Aspect Technologies (SPLAT'03)*, Boston, Massachusetts, April 2003.