Design-Based Pointcuts Robustness Against Software Evolution

Walter Cazzola[1], Sonia Pini[2], and Ancona Massimo[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|ancona}@disi.unige.it

Abstract. Aspect-Oriented Programming (AOP) is a powerful technique to better modularize object-oriented programs by introducing crosscutting concerns in a safe and noninvasive way. Unfortunately, most of the current join point models are too coupled with the application code. This fact harms the evolvability of the program, hinders the concerns selection and reduces the aspect reusability. To overcome this problem is an hot topic.

This work propose a possible solution to the limits of the current aspect-oriented techniques based on modeling the join point selection mechanism at a higher level of abstraction to decoupling base program and aspects.

In this paper, we will present by examples a novel join point model based on design models (e.g., expressed through UML diagrams). Design models provide a high-level view on the application structure and behavior decoupled by base program. A design oriented join point model will render aspect definition more robust against base program evolution, reusable and independent of the base program.

1 Introduction

Aspect-oriented programming (AOP) is a powerful technique to better modularize object-oriented programs by introducing crosscutting concerns in a safe and noninvasive way. Each AOP approach is characterized by a *join point model* (JPM) consisting of the *join point*s, a means of identifying the join points (*pointcut*s) and a means of raising effects at the join points (*advice*). Crosscutting concerns may not be well modularized as aspects without an appropriate join point definition that covers all the interested elements, and a *pointcut definition language* that allows the programmer of selecting them.

Traditionally, the pointcuts allow the programmer of selecting the join points on the basis of the program lexical structure, such as explicit program elements names. The dependency on the program syntax renders fragile the pointcuts definition [2, 11] and strictly couples an aspect to a specific program harming the evolvability [15] and hindering the aspect reusability [7].

At the moment, aspects are not robust against evolutions in the base program. This is because pointcut definitions typically rely heavily on the structure of the base program. This tight coupling of the pointcut definitions to the base program's structure can seriously hinder the software evolution. Thus, this implies that all pointcuts of each aspect need to be checked and possibly revised whenever the base program evolves.

To get the *obliviousness* [3] the aspect programmer should be unaware of the structure and syntax of the base-level program to apply its aspects as well as the base-level programmer must be unaware of the additional aspects. To get a total obliviousness[3] means also to decouple the aspect definitions from the dependency on the structure and syntax of the program they advice, solving the abovementioned problems.

Therefore, the required enhancement should consist of developing a pointcut definition language that supports join points selection on a more semantic way. To provide a more expressive and semantic-oriented selection mechanism means to use a language that captures the base-level program behavior and properties abstracting from the syntactic details. Several attempts in this direction have been done but none of these really approaches the problem in its entireness and in general they raise also new issues, such as efficiency and flexibility. We think that the *design models* provides a more suitable representation to abstract join points identification from the base-code structure and syntax.

In this paper, we propose a design oriented join point model that should offer the right level of abstraction from the base-code. In particular, in our proposal, join points are described by means of UML-like descriptions (basically, activity and sequence diagrams) representing computational patterns, these elements are called *join point patterns*. In other word, we propose of using enriched UML diagrams (or portion of) to describe the control flows or the computational contexts and the join points inside these contexts to detect possible woven points. Pointcuts consist of logic composition of join point patterns. In this way, pointcuts are not tailored on the program syntax and structure but they are more general.

The rest of the paper is organized as follows: in section 2 we overview the limitations against the software evolution of the AspectJ-like join point models, in section 3 we introduce our join point model and in particular the concept of *join point pattern*, finally, in section 4 and in section 5 we face some related works and draw out our conclusions.

2 Limits of the AspectJ-Like JPM against Software Evolution

The join point model has a critical role in the applicability of the aspect-oriented methodology. As stated by Kiczales in his keynote at AOSD 2003 [9] the pointcut definition language has the most relevant role in the success of the aspect-oriented technology.

Most of the AOP approaches use a join point model similar to that of AspectJ [10]. It exploits a dynamic call graph [6] to select the correct join points. 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. These primitive pointcut

[3] As *total obliviousness*, we mean the unawareness of the base-level program of the existence of the aspects and vice versa.

designators can be combined using logical operations (`||`, `&&`, `!`) forming more complex pointcuts. 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.

Therefore, most AOP approaches have a tight coupling between aspects and base program, even if the aspect definition is syntactically separated from the base program, changes to the base program can immediately require changes to the aspect definition. 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. This is in direct contrast with the general aim of AOP, that is, to make programs easy to read, manage and evolve, by providing new modularization mechanism.

Pointcut heavily relies on how the software is structured at a given moment in time. In fact, the aspect developer subsumes the structure of the base program when he/she defines the pointcuts; the name conventions are an example of this subsumption. The 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 and obliviousness 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*. For example, the following `setterAccess()` pointcut should capture all the methods that modify the state of the object.

```
pointcut setterMethod() : call(* set*(..));
```

To define this semantic property, the pointcut relies on the coding convention that the name of this kind of methods *always* starts with the prefix `set`. Since the rule subsumed by this pointcut is not imposed by any mechanism, not all developers need to be aware of its existence and, consequently, of having to respect it; in practice this rule gets broken very often. During the base program evolution new methods can be added and existing ones can be removed such that they are captured by the pointcut definition only if they follow the naming convention.

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.

3 Design-Based Pointcut Language

Design models (UML diagrams, formal techniques and so on) provide the right level of abstraction necessary to have a global and static view of the system and to select the join points thanks to their properties and where they are located (i.e., the context) [2], and then to obtain a more robust pointcut mechanism against the software evolution. We

Pattern-Based Join Point Model: Terminology and Element Description
`Join Points` *They are hooks where code may be added. We consider two different kind of join point, normal join points that represent points of the application behavior where to insert the advice code, and around join points that represent portion of application behavior that must be substituted with the advice code. They are pointed out by one or more join point patterns and refer to the application code.*
`Join Point Patterns` *They are UML diagrams, with a name, that describe a set of join points in terms of their application context. These patterns provide an incomplete and parametric representation of the application behavior. The set of all the declared join point patterns is called the join point pattern space.*
`Join Point Pattern Space` *It is the set of all join point patterns defined into the application.*
`Pointcut` *It is a query on the join point patterns space selecting a set of join points. The queries are created as logic composition of the join point names identified into the join point pattern space.*
`Advice` *It is the code applied at the join points when the associated pointcut is evaluated to true.*

Table 1. Pattern-Based Join Point Model: Terminology and Description

propose to tackle the join point model problems by selecting the join points in terms of the base program design models.

Model-based pointcut definitions are less subject to the *fragile pointcut problem* [11], and then they are more robust against evolution problems, because they are not defined in terms of how the program is structured at a certain point in time. Since, model-based pointcut definitions are decoupled from the structure and syntax of the base program, the fragile pointcut problem is transferred to a more conceptual level. By defining pointcuts in terms of a design model, the fragile pointcut problem has now been translated into the problem of keeping the right localization of the design context and the join points into the base program.

The pointcut definition mechanism we are proposing, called *join point pattern specification language* selects the join points in terms of the base program design models. The application design models provide an abstraction over the application structure. Thanks to this abstraction, the join point patterns can describe the join point position in terms of the application behavior rather than its structure. In other words, we achieve a low coupling of the pointcut definitions with the source code since the join point pattern definition is defined in terms of design model rather than directly referring to the implementation structure of the base program itself.

The join point patterns are graphically specified through a UML-like description — sequence and activity diagrams. A visual approach is more clear and intuitive and makes more evident the separation from the program source code. Finally, UML-like approach is not limited to a specific programming language but can be used in combination with many. At the moment, we are using the `Poseidon4UML` program for depicting the join point patterns but we are developing an ad hoc interface for that.

In general, software evolution involves both structural (e.g., add classes, methods, fields and so on) and behavioral changes, then the pointcuts can affect both the structure and the behavior. In this paper, we only focus on the behavioral join point pattern

```
abstract aspect Observer {
  void notify() { ... }
  abstract pointcut p();
  abstract pointcut c();
  after(): p() {notify();}
  after(): c() {notify();}
}
```

```
aspect Observing1 extends Observer {
  pointcut p(): call(void Buffer.put(int));
  pointcut c(): call(void Buffer.get());
}
```

```
aspect Observing2 extends Observer {
  pointcut p():within(Buffer) && call(* put*(*);
  pointcut c():within(Buffer) && call(* get*(*));
}
```

Fig. 1. The abstract observer pattern aspect with two concrete implementations.

definition; since affecting the application structure simply consists on introducing and removing elements and can be faced as explained in [1].

3.1 The Join Point Pattern Specification Language

In this paper, we borrowed the terms *join point* and *pointcut* from the AspectJ terminology but we use them with a slightly different meaning. The *join point*s are *hooks where code may be added* rather than *well defined points in the execution of a program*. Whereas, the *poitcut*s refer to a set of join points. To complete the picture of the situation, we have introduced a new concept: the *join point pattern* as *a template on the application behavior identifying the join points in their context*. These patterns provide an incomplete and parametric representation of the application behavior. Look at Table1 for a summary and brief description of the elements composing the model.

In addition to decoupling the pointcut definitions from the base code, design-based join point patterns are less fragile to evolution of the base program because the pointcut definitions are based on composition of join points, that are no-linked to the application structure and syntax but linked to the behavior of the application.

A join point pattern is a sample of the computational flow described by using a behavioral/execution flow template. The sample does not completely define the computational flow but only the portions relevant for the selection of the join points. The set of all defined join point patterns is called *join point pattern space*. Each join point pattern can describe and capture many join points; these join points are captured together but separately advised. Pointcuts are expressed as a logic combination of one or more join point patterns.

Now, we will explain the join point pattern definition language "syntax" by examples. Let us consider the implementation of the *observer pattern* [4] as an aspect to observe the state of a *buffer*. The `Buffer` instances originally support only two kinds of operations: to retrieve (`get`) and to insert (`put`) elements in the buffer. The observer will monitor the work of these tow family of methods.

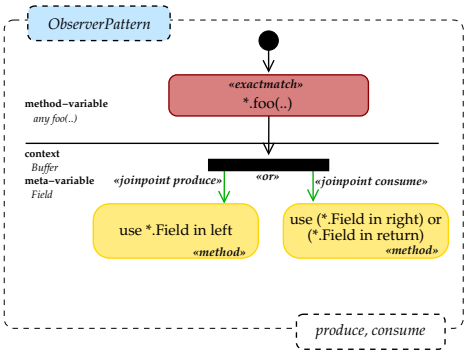

Fig. 2. A Join Point Pattern capturing all the state changes in the `Buffer` class.

Figure 1 shows an abstract aspect (written in AspectJ) that implements the observer pattern behavior with two possible concrete implementation of its pointcuts. The use of an abstract aspect is a way to decouple the crosscut definition from the aspect. The first concrete aspect is based on enumerating the method calls of the base-program, whereas the second one is based on the use of name conventions and wildcards. Both these concrete aspects capture all the interested join points in the case of a buffer implementation which respects the implicit programming conventions proposed from the problem statement, but what happens when the buffer class evolves in a way that violates the self-imposed programming conventions?

To answer to this question, we consider few possible evolutions of the `Buffer` class. First case, we add a method `"void putAll(int [])"` to the `Buffer` class. This event breaks the first concrete aspect because the new method is not listed in the `p()` pointcut. To maintain the expected behavior of the aspect, the pointcut must be modified to include also the new method. The second concrete aspect is more robust and the new method is automatically captured by it because it respects the naming conventions and start by `put`.

Now, let us consider a new change: a method `returnElements` is added. This new method returns a collection with a specified number of elements from the buffer. In this case, both first and second concrete aspect do not capture the join points introduced by calling the new method. The first for the same reason raised in the previous example and the second since the name of the new method does not respect the self-imposed conventions.

Figure. 2 shows a join point pattern capturing *all the method executions which change the state of the* `Buffer` *class*, i.e., our join point pattern can capture both the executions to methods that retrieve data from the buffer and that introduce data in the buffer.

The behavior we are looking for is characterized by: i) the call to a method with any signature, ii) whose body either assign anything to a field of the target object (to select the `put` method family) *or*, either assign a field of the target object to anything or return a field of the target object (to select the `get` method family). This join point

pattern explicitly refers to the concept of a method that change the `Buffer` state rather than trying to capture that concept by relying on implicit rules about the program implementation structure. Consequently, the pointcut defined using this pattern does not need to be verified or changed to be compliant with the evolution of the base program: if the context of the pattern correctly classifies all methods which change `Buffer` state, the pointcut remains correct. By using our `ObserverPattern` the new `putAll(int[])`, and `returnElements(int)` methods will be automatically captured.

The activity diagram describes the context where the join points could be found, more details are used to describe the context and more the join point pattern is coupled to the application code. The use of *meta variables* grants the join point pattern independence from a specific case. In the example, `foo` and `Field` are meta-variables, respectively a *method meta-variable*, i.e., a variable representing a method name and a *variable meta-variable*, i.e., a variable representing a variable name. In this example the method signature is not specified, therefore any method call could be captured if it has the right behavior independently of its signature. If necessary, type meta-variable, i.e., a variable whose values range on types, can be used to define the method signature. Meta-variables got a value during the pointcut evaluation and their values can also be used by the advice.

In the caller swimlane[4], we look for the invocation of the `foo(..)`[5] method whereas in the callee swimlane we look at the method body for either the assignment to a generic class field or, either the use of generic class field into the right of an assignment or the use of the field in a return statement. The former should be an exact statement match, — i.e., we are looking for exactly that call — whereas in the latter we are looking for a specific use of a field in the whole method body. This difference can be expressed by using the join point pattern syntax and a couple of stereotypes:

- a rounded rectangle, called *template action*, indicates that we are looking for the use of a meta-variable in the next statements, a stereotype set a constraint for the searching scope; «`method`» limits to the method body whereas «`block`» limits to the current block;
- we can look for the use of a meta-variable in a left (`left`) or right (`right`) part of an assignment, in a boolean expression (`booleanCondition`), in a generic statement (`statement`), and in a return statement or in their logic combination;
- a rounded rectangle with the «`exactmatch`» stereotype, called (according to UML) *action*, indicates one or more instructions, expressed following the Java syntax; the names used inside this block can be either meta-variables, constant variable names or if not useful to the pattern definition indicated as `(i)` with $\texttt{i} \in \mathbb{N}$.

The join point possible location is indicated by the «`joinpoint`» stereotype attached to an arrow. Each join point pattern can describe the context for many join points that can be located by using a «`joinpoint`» stereotype with a different name. All the captured join points are listed in the window in the low-right corner of the join point pattern specification. In Fig. 2 we have two different join points called respectively `produce` and `consume`.

[4] A swimlane is a way to group activities performed by the same actor/object.

[5] Please note that `foo(..)` is meta-variable and method signature is not specified.

We have adopted a loose approach to the description of the computational flow. In the join point pattern, based on activity diagrams, the lines with a solid arrowhead connecting two elements express that the first one follows immediately the other, and the lines with a stick arrowhead (see Fig.2) express that the first follows the other, but not immediately, i.e., zero or more *not relevant* actions[6] could happen before the second action, the number of actions that could happen is limited by the scope.

Our join point model is strictly based on the structure of the computational flow, so we don't need to differentiate between `before` and **`after`** advice but we can simply attach the «`joinpoint`» stereotype in the right position, i.e., before or after the point we would like to advice. A special case is represented by the *around join point pattern*s which match portions of the behavior instead of a single point; the whole matched portion represents the join point and will be substituted by the advice code.

3.2 Aspects that Use Join Point Patterns

The showed join point pattern simply describes where the join points can be found, to complete the process we must declare an aspect where the join point pattern is used to associate the advice code at the interested join points.

The aspect definition, like in most AOP languages, includes pointcuts definition and advices linked to these pointcuts. Moreover, the aspect must declare all the join point patterns it uses and which join points it imports from them. Both pointcuts and advices will use these information in their definition.

The following `Observer` aspect imports the `produce` and the `consume` join points from the `ObserverPattern` join point pattern.

```
public aspect Observer {
    void notify() { ... }
    public joinpointpattern ObserverPattern(produce, consume);
    public pointcut p(): produce();
    public pointcut c(): consume();
    advice() : p() {notify();}
    advice() : c() {notify();}
}
```

4 Related Works

This paper propose to decoupling base programs and aspects using an UML-based join point model by approaching the join points selection on a less syntactical and structural basis. To get a semantic join point model to avoid the fragile pointcut problem is a quite hot topic and several approaches are currently under investigations.

In [13], Noguera et al. present a mechanism to express type-safe source code templates in pure Java that improves the expressiveness of pointcut languages. To have a more semantic pointcut language, they propose to match, not only on the signature, but

[6] 3These actions do not participate in the description of the join point position, so they are considered not relevant.

also on the structure of the method. They propose a way to extend AspectJ pointcut language with structural constructs in the form of typesafe native Java source code templates, where templates, define a source code model in which some elements are variable. The basic idea is similar, i.e., identify join points not only on the base of method signature but also on method behavior.

In [7] Kellens et al. propose a novel technique of model-based pointcuts, which translates the fragile pointcut problem to a more conceptual level where it is easier to solve. This is done by decoupling the pointcut definitions from the actual structure of the base program, and defining them in terms of a conceptual model of the software instead.

In [5] Gybels et al. present a logic-based crosscut language, called CARMA. The use of a crosscut language based on logic programming it gets the use of unification as a more advanced wildcard mechanism, the use of logic rules for writing reusable pointcut.

In [8] Kellens et al. present a method for keeping the conceptual model documentation consistent with the source code when the program evolves. In particular they implement a particular solution to the fragile pointcut problem through an extension of the CARMA aspect language combined whit the formalism of intensional views [12].

Pointcut delta analysis [14] tackles the fragile pointcut problem by analyzing the difference in captured join points, for each pointcut definition, before and after an evolution. Their approach to deal with the fragile pointcut problem for current languages.

Although such expressive pointcut languages permit to render pointcut definitions much less fragile, but none of these languages approaches the problem in its entireness. A pointcut definition still needs to refer to specific base program structure or behavior to specify its join points. This dependency on the base program remains an important source of fragility.

5 Conclusions

Current AOP approaches suffer from well known problems that rely on the syntactic coupling established between the application and the aspects. This is a serious inhibitor to evolution of aspect-oriented programs. A common attempt to give a solution consists of freeing the pointcut definition language from these limitations by describing the join points in a more semantic way.

This paper shows the robustness against evolution of a design-based approach to join points identification. This approach allows of decoupling aspects definition and base-code syntax and structure, and of rendering the pointcut definitions less fragile against the base program evolution. Pointcuts are specified using UML-based join point pattern. More precisely, a join point pattern is a template on the application behavior identifying the join points in their context. In particular join points are captured when the pattern matches portion of the application behavior.

Compared with current approaches, we can observe some advantages; first of all, we have a pointcuts definition more behavioral. In the join point pattern definition we identify the context of the computational flow we want to match, and precise point we want to capture, weaken the coupling of the aspect to the base program and hence,

providing crosscuts that are more robust towards evolution. The graphical definition of join point patterns is more intuitively and comprehensible for programmers. Moreover, a visual view of the context in which locate the join points would be preferred since it better demonstrates where and how an aspect can influence a program.

References

1. Walter Cazzola, Antonio Cicchetti, and Alfonso Pierantonio. Towards a Model-Driven Join Point Model. In *Proceedings of the 11th Annual ACM Symposium on Applied Computing (SAC'06)*, pages 1306–1307, Dijon, France, on 23rd-27th of April 2006. ACM Press.
2. 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.
3. 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.
4. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, Ma, USA, 1995.
5. 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.
6. Erik Hilsdale and Jim Hugunin. Advice Weaving in AspectJ. In *Proceedings of the 3rd Int'l Conf. on Aspect-Oriented Software Development (AOSD'04)*, pages 26–35, Lancaster, UK, March 2004. ACM Press.
7. 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.
8. Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*, Nantes, France, July 2006. Springer.
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. Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. Co-evolving Code and Design Using Intensional Views - A Case Study. *Journal of Computer Languages, Systems and Structures*, 32(2):140–156, July/October 2006.
13. Carlos Noguera and Renaud Pauwlak. Open Static Pointcuts Through Source Code Templates. In *Proceedings of Open and Dynamic Aspect Languages Workshop (ODAL'06)*, Bonn, Germany, March 2006.
14. 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.

15. Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Chai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. On the Criteria to be Used in Decomposing Systems into Aspects. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*, Lisbon, Portugal, September 2005.