Non-invasive and Non-scattered Annotations for More Robust Pointcuts

Leonardo Silva, Samuel Domingues, Marco Tulio Valente
Institute of Informatics, PUC Minas, Brazil
{leonardosilva,mtov}@pucminas.br, samueldro@gmail.com

Abstract

Annotations are often mentioned as a potential alternative to tackle the fragile nature of AspectJ pointcuts. However, annotations themselves can be considered crosscutting elements because they are normally pervasive and tangled with business-specific functionality. In this paper, we propose a solution to the fragile pointcut problem in aspect-oriented programming that relies on non-invasive and non-scattered annotations. The central components of the proposed solution are so-called annotator aspects, that superimpose annotations to the base code in a non-invasive way. Moreover, annotator aspects are generated semi-automatically, from a declarative annotation definition language. The paper presents examples of using the proposed solution in pointcut descriptors of two real-world aspect-oriented systems. We also describe a case study that evaluates the robustness of the proposed solution in face of possible changes to the classical Figure Editor system.

1 Introduction

Aspect-oriented programming (AOP) extends traditional programming paradigms with powerful abstractions targeting the modularization of crosscutting concerns. AspectJ is considered nowadays one of the most stable aspect-oriented languages. AspectJ extends Java with new modularization abstractions, including join points, pointcuts, advices, and aspects. In AspectJ, an *aspect* defines a set of *pointcut* descriptors that matches well-defined points in program execution (called *join points*). Advices are anonymous methods that are implicitly invoked before, after or around join points captured by pointcuts. Aspects are supposed to promote reusability, maintainability and comprehensibility, since the implementation of the otherwise spread and tangled code related to crosscutting concerns is modularized in a single component.

Surprisingly for a paradigm with sophisticated modularization abstractions, aspects as implemented by AspectJ can hamper program evolution, leading to what has already been called the *AOSD-Evolution Paradox* [17]. The origin of this paradox is that pointcuts rely on names and wildcards to capture join points over programs written by oblivious programmers (i.e. programmers that do not know the existence of aspects). As a result, changes in the structure or in the naming conventions of the base program can silently trigger the execution of unwanted advices or prevent the execution of required ones. In other words, the current pointcut model employed by AspectJ implies in a tight coupling between pointcut descriptors and the structure of the base program, although such coupling is not explicitly documented and checked by the compiler or weaver. This problem has been studied by other researchers, usually under the name of *fragile pointcut problem* [15, 13, 7].

In this paper, we rely on previous research that proposes the use of annotations to design more robust pointcuts [10, 3, 6]. Particularly, we cope with the following problems that are typical when combining aspects and annotations:

- *Annotations are crosscutting concerns.* Java annotations present a crosscutting behavior because they are normally pervasive and tangled with business-specific functionality. In order to tackle this problem, we propose the use of so-called annotator aspects, that superimpose annotations to the base code in a non-invasive way. Moreover, we propose that annotator aspects should be generated semi-automatically, from a declarative language.

- *Annotation-based pointcuts are fragile.* Traditional annotation-based pointcuts presume that maintainers have annotated the base program correctly. Particularly, missing annotations or annotations that do not follow implicit naming conventions can silently disable join points that should have been captured by annotation-based pointcuts. In order to cope with this problem, our solution explicitly requires maintainers to inform whether each target element in the lexical range of an annotation should be annotated or not. From this information, the system generates the annotator aspects.

978-1-4244-2614-0/08/$25.00 © 2008 IEEE

- *Annotations hamper obliviousness.* Although usually considered as a distinguished property of AOP, the idea of obliviousness has been the subject of earlier revision. For example, Sullivan et. al have demonstrated that obliviousness in its purest form leads to pointcuts that are hard to develop, understand and evolve [4, 16]. For this reason, they recommend that developers should first prepare base programs to aspects. Particularly, in the solution proposed in this paper, developers are forced to decide whether base program elements should be annotated with previously declared annotations. Thus, annotations can be considered as contracts (or design rules) between aspects and base programs.

The remaining of the paper is organized as follows. Section 2 motivates the need for more robust pointcut descriptors using the classical Figure Editor system common in AOP papers. Section 3 presents the solution proposed in this paper and its main components, including a annotation declaration language (used to declare annotations), annotation-aware interfaces (used to document which annotation applies to which member of the classes of the base program), and annotator aspects (used to annotate base program elements in a non-invasive way). Section 4 presents examples of using the proposed solution in pointcut descriptors of real-world aspect-oriented systems. Section 5 describes a case study that evaluates the robustness of the proposed solution in face of possible changes to the Figure Editor system. Section 6 presents a discussion about the proposed solution, outlining its strengths and limitations. Section 7 discusses related work, and Section 8 concludes.

2 Fragile Pointcut Problem

The fragile pointcut problem is analogous to the fragile base class problem in object-oriented programming [11]. In OO development, developers cannot determine whether a base class change is safe simply by examining its methods in isolation. Instead, they also need to examine the methods of the subclasses as well. Translating the problem to aspects, in order to determine whether a base program change is safe developers must examine possible impacts in the join point shadows captured by the pointcuts declared in the program. Particularly, a pointcut is considered fragile (or not robust) when even trivial modifications in the base program can silently lead to one of the following situations: *unintended capture of join points* or *accidental join point misses* [7].

The classical Figure Editor system presented in Figure 1 is used to illustrate both situations. For this system, suppose an advice that refreshes the display whenever figures change (Figure 2a). Furthermore, suppose alternative implementations for the `change` pointcut based on regular expressions, enumerations, and annotations (Figure 2b to 2d). We show next that such implementations are fragile with respect to evolutions in the base program.

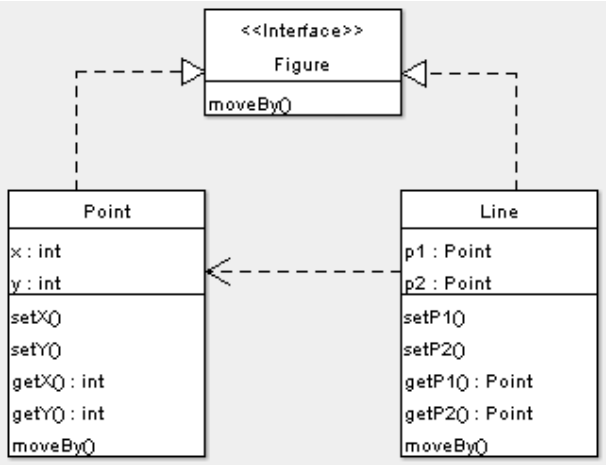

Figure 1. Motivating Program

```
(a) after() returning: change() {
      Display.refresh();
    }

(b) pointcut change():
      execution(void Figure+.set*(..));

(c) pointcut change():
      execution(void Point.setX(..)) ||
      execution(void Point.setY(..)) ||
      execution(void Line.setP1(..)) ||
      execution(void Line.setP2(..));

(d) pointcut change():
      execution(void @DisplayStateChange *.*(..))
```

Figure 2. Refresh display advice and three alternative implementations for its change pointcut

Regular expression based pointcuts: Regular expressions are subjected both to unintended capture of join points or to accidental join point misses. For example, suppose a change that requires adding a `date` field to the class `Point` (representing the last time the figure was saved on persistent store) and an associated `setDate` method. In this case, executions of `setDate` will be captured by the `change` pointcut, which constitutes an unintended capture of join point. On the other hand, suppose that base program developers decide to rename method `setX` to `changeX`. In this case, executions of the renamed method will no longer be captured by the `change` pointcut and the

affected `Point` object will not be refreshed in the display. Thus, this last change constitutes an accidental join point miss.

Enumeration-based pointcuts: Enumeration-based pointcuts are particularly fragile to join point misses. For example, extending the previous enumeration is needed whenever new methods that change the display state are included in the base program.

Annotation-based pointcuts: Such pointcuts presume that base program developers and maintainers have correctly annotated the methods that change the display. However, this assumption can be easily violated by oblivious developers during the evolution of the program, leading to join point misses. Furthermore, as described in Section 1, annotations usually present a crosscutting behavior.

3 Proposed Solution

Figure 3 describes the main components of the proposed system. First, developers must declare the annotations employed in the pointcuts of an aspect-oriented system using an annotation declaration language. The solution also includes a tool that queries users whether particular program elements should be annotated or not. This tool, called annotator, is executed before the weaver. The annotator is responsible for the creation of two other key components of the system: annotator aspects and annotation-aware interfaces. These components are described in details in the remaining of this section.

3.1 Annotation Declaration Language

The proposed solution supposes that annotations are declared using an Annotation Declaration Language. Programs in such language consist of entries according to the grammar described in Figure 4. In this grammar, non-terminal symbols start with capital letters; terminals start with non-capital letters; $\{A\}$ denotes zero or more repetitions of A; and $[A]$ denotes that A is optional.

The declaration of an annotation includes the following information:

- The name of the annotation (entry `name`).
- The target of an annotation (entry `target`). Annotations can be associated to fields, methods, and classes. In case of methods, developers can restrict the annotation to methods that are `getters` (i.e. methods that just return a field), methods that are `setters` (i.e. methods that just set fields), methods whose body contain a `call` to a defined method, methods that `declare` local variables or formal parameters of a given type, and methods that contain a `catch` for exceptions of a given type. In case of annotations associated to classes, developers can restrict the annotation to classes that are used to declare fields, return types, method parameters or exceptions. Annotations can also be restricted to classes that include a `field` of a given type.
- The lexical scope of an annotation is the enclosing program unit to consider when looking for possible targets of an annotation. The scope of an annotation can be a class (and its subclasses), classes that implement a given interface or classes declared in a given package.
- The question that should be presented to developers so that they can decide whether a candidate target element should be annotated or not (entry `question`).

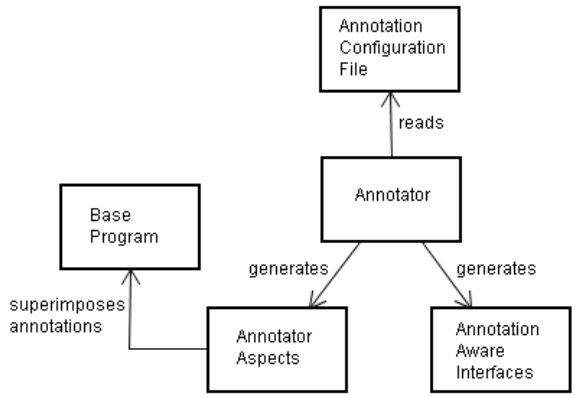

Figure 3. Components of the proposed system

```
AnnotationDeclarationFile ::=
   { annotation AnnotationSection end }

AnnotationSection ::=
    name     = String
    target   = fields
             | methods [&& MethQualifier]
             | classes [&& ClassQualifier]
    scope    = Type | Package
    question = String

MethQualifier ::= getters | setters
                  call(Method) |
                  declare(Type) |
                  catch(Type)

ClassQualifier ::=
      usedAs(field|param|return|exception) |
      field(Type)
```

Figure 4. Annotation Declaration Language

The proposed solution explicitly requires developers to declare annotations, providing information about their target elements and scope. When defining the target of an annotation, developers can also provide structural information about such elements, in order to reduce the number of considered targets. Based on this information, the annotator component monitors the codebase after each compilation, searching for program elements that match the declared annotations. After detecting a potential target, the annotator prompts the developer to confirm if it should be annotated or not. In case of an affirmative answer, the annotator generates code to introduce the annotation. This code is generated in a so called annotator aspect, thus avoiding annotation scattering and tangling. Moreover, by reminding developers to decide whether a matched target element should be annotated or not, it handles the join point miss problem that is common when using standard Java annotations.

The following entry illustrates the specification of the `DisplayStateChange` annotation (as required by the `change` pointcut of Figure 2d):

```
annotation
   name= @DisplayStateChange
   question= Does method %target change the state of
             the display
   target= methods && (!getters)
   scope= Figure
end
```

This entry defines that potential targets of `DisplayStateChange` are methods that are not getters and that are defined in classes that implement the `Figure` interface[1]. The annotator relies on this information to search for methods in such classes matching the defined annotations. The `question` entry in the declaration of an annotation specifies the question used by the annotator to inquiry developers about a new identified target element (Figure 5). In other words, the system detects methods that can potentially change the state of a `Figure` and delegates to developers the decision whether such change should require refreshing the display or not.

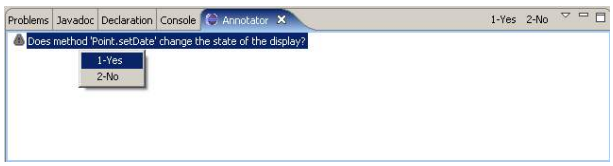

Figure 5. Prompting developers about a new program element

It is important to mention that the system constantly observes the base program for possible annotation targets. For example, maybe a method `m1` does not attend the declaration of an annotation `ann` when it is first implemented (for example, because `ann` only applies to methods that call another method `m2`). Afterwards, changes in `m1` may insert a call to `m2` to its implementation, thus turning it into a candidate to the annotation `ann`.

[1]For the sake of clarity, we have omitted from the previous grammar the fact that qualifiers (such as getters) can be used in boolean expressions.

3.2 Annotation-Aware Interfaces

Annotation-aware interfaces extend traditional interfaces with information about their annotated elements. Essentially, each class of the base program has an annotation-aware interface, which is generated automatically by the annotator tool. Such interfaces list fields and methods of the associated class that have been annotated. The annotation-aware interfaces associated to the classes of the motivating program are the following:

```
class Point {
  @DisplayStateChange void setY(int);
  @DisplayStateChange void setX(int);
}
class Line {
  @DisplayStateChange void setP1(Point);
  @DisplayStateChange void setP2(Point);
}
```

Annotation-aware interfaces are a variant of aspect-aware interfaces (AAI), as proposed by Kiczales and Mezini to support modular reasoning in aspect-oriented systems [9]. An AAI lists the fields and methods of a class and the advices that are implicitly called when the listed elements are accessed. Instead of providing information about advices, the proposed annotation-aware interfaces exhibit the annotations associated to base program elements. Exhibiting annotations helps modular reasoning because maintainers can avoid changes in the base program that invalidate semantic properties related to annotations. Moreover, maintainers can directly edit annotation-aware interfaces to remove or to add annotations. For example, this may be necessary when they realize that have answered wrongly the question to attach an annotation to a given program element. By editing annotation-aware interfaces they can remove an incorrect annotation associated to a given target element or add an annotation to a given program element.

3.3 Annotator Aspects

Each declared annotation has a corresponding annotator aspect, that superimposes the annotation to the base program in a non-invasive way. This aspect is generated (and updated) semi-automatically by the annotator tool whenever a new target element is identified in the lexical scope of an annotation, and the developer confirms that it should be annotated. Moreover, annotator aspects are also updated when

developers edit the annotations listed in the annotation-aware interfaces of the system.

The following annotator aspect illustrates the superimposition of the `DisplayStateChange` annotation in our motivating example:

```
aspect DisplayStateChange {
 declare @method:
    public void Point.setX(int): @DisplayStateChange;
 declare @method:
    public void Point.setY(int): @DisplayStateChange;
 declare @method:
    public void Line.setP1(Point): @DisplayStateChange;
 declare @method:
    public void Line.setP2(Point): @DisplayStateChange;
}
```

4 Examples

4.1 JAccounting

JAccounting[2] is a Web-based business accounting system that relies on the well-known Hibernate framework for persistence and transaction services. Binkley et. al have refactored the original JAccounting implementation in order to use aspects to modularize transaction management [1]. For example, the refactored system uses the following aspect to modularize the code in charge of starting a transaction:

```
aspect Transaction {
  Transaction tx;
  pointcut p_0():
     call(Session SessionFactory.openSession() &&
     (( withincode(String ProductsPage.perform2())
     || withincode(String InvoicePage.perform2())
     || withincode(String RecurrencePage.perform2())
     || withincode(String PaymentPage.perform2())
     || withincode(String CustomerDetails.perform2())
     || withincode(String CustomerForm.perform2()
     || withincode(Account createInternalAccount(
        Session, Integer, int));
  after() returning (Session sess)
          throws HibernateException: p_0() {
    tx= sess.beginTransaction();
  }
}
```

Pointcut `p_0` defines the join points that require starting a transaction context. Such points correspond to calls to the `openSession` method occurring in the lexical scope of the methods enumerated in the pointcut descriptor. This type of pointcut implementation can easily lead to failures in starting a transaction, when maintainers do not update `p_0` with new methods requiring transaction handling.

Alternatively, using the solution proposed in this paper, a `Transactional` annotation is declared in the following way:

[2] `https://jaccounting.dev.java.net`.

```
annotation
  name= @Transactional
  question= Is method %target transactional
  target= methods && call(Session.save)
end
```

The target of this annotation are methods whose implementations statically include calls to the method `save` from class `Session`, which represents the session with the database. Clearly, only such methods are candidates to transaction handling in Hibernate-based systems. As described, the annotator will ask developers to confirm if such methods actually demand transaction handling or not. In fact, JAccounting has eleven methods that call `Session.save`. From those, seven methods require starting a transaction (as enumerated in pointcut `p_0`). Thus, for the remaining methods, developers must answer negatively to the question formulated by the annotator. Although they update the database, such methods do not require transaction handling for various reasons, including for example the fact that they share the session object with another transactional method.

Using the declared annotation, developers in charge of implementing transaction handling can rely on the following pointcut descriptor:

```
pointcut p_0():
  call(Session SessionFactory.openSession()) &&
  (withincode(@Transactional * *(..));
```

In case JAccounting evolves and new transaction methods are implemented, this pointcut remains valid. In such scenarios, developers must only confirm that the new detected methods (i.e. methods that save data in persistent store) demand transaction handling code.

4.2 HealthWatcher

HealthWatcher is a Web-based information system used by citizens to register complaints about the sanitary conditions of restaurants and food shops. A first experience of using AspectJ to modularize distribution and persistence concerns of HealthWatcher has been conducted by Soares, Borba and Laureano [14]. At least, two aspects of the system can benefit from the solution described in this paper: persistence and transactional control.

Persistence: In the aspect-oriented version of the system, the following pointcut is used to match calls to methods that require synchronizing the target object with the database:

```
pointcut remoteUpdate(PersistentObject o):
   this(HttpServlet) && target(o) && call(* set*(..));
```

The tag interface `PersistentObject` is used to identify classes whose objects must be updated after being changed. Such classes are marked by intertype declarations as the following one:

```
declare parents: Complaint implements PersistentObject;
```

Similar intertype declarations are defined for other classes that demand data synchronization, such as `Employee` and `HealthUnit`. In case HealthWatcher evolves and new persistent classes are implemented, developers must update the mentioned intertype declarations. Alternatively, using the solution proposed in this paper, a `Persistent` annotation is declared in the following way:

```
annotation
  name= @Persistent
  question= Is class %target persistent
  target= classes && usedAs(param)
  scope= HealthWatcherFacade
end
```

The target of this annotation are classes used as parameter types in methods of the system facade class (`HealthWatcherFacade`). Such types are natural candidates to persistence, since the system architecture defines that the facade is the unique entry point of the system, used by different clients to access and update HealthWatcher business collections objects. As usual, in order to avoid false positives, the annotator will ask developers to confirm whether the matched classes actually demand persistence or not. In fact, there are three types used as parameters in the methods of `HealthWatcherFacade`. All of them represent persistent objects.

Using the declared annotation, data state synchronization aspects can rely on the following intertype declaration:

```
declare parents:
   (@Persistent *) implements PersistentObject;
```

This declaration prescribes that any class annotated as `Persistent` must implement the `PersistentObject` interface. When the system evolves and new persistent classes are implemented, maintainers do not need to update manually the list of intertype declarations anymore.

Transactional Control: In the AO version of HealthWatcher, transactional methods should be explicitly declared in an interface named `ITransactionalMethods`. This interface is used by the following pointcut to capture the execution of methods that require transactional control:

```
pointcut transactionalMethods():
  execution(* ITransactionalMethods.*(..));
```

The main drawback of this solution is that developers and maintainers are oblivious to the existence of the interface `ITransactionalMethods`. Thus, new transactional methods will be added to the system without updating this interface, leading to join point misses. In order to tackle this problem, the following `Transaction` annotation can be employed:

```
annotation
  name= @Transactional
  question= Is method %target transactional
  target= methods
  scope= HealthWatcherFacade
end
```

The declaration of this annotation requires developers to classify methods in the facade of the system – including methods added in future versions – according to their transactional behavior. It is worth to mention that HealthWatcher architecture prescribes that only such methods are subject to transactional control. Currently, the system facade has 15 methods, from which five are transactional.

Using the declared annotation, the following pointcut can be employed to capture the execution of methods that require transactional control:

```
pointcut transactionalMethods():
  execution(@Transactional * * (..));
```

This pointcut is fully decoupled from the abstract syntax of the base program. Moreover, the proposed solution automatically reminds developers about new methods added to the system facade. Developers should then decide if they want to execute such methods in a transactional context or not.

5 Robustness Study

In order to evaluate the robustness of the pointcuts engendered by our solution, this section relies on an adaptation of the change scenarios proposed by Ostermann, Mezini and Bockisch to the classical Figure Editor system [13]. We have evaluated the impact of the proposed change scenarios regarding four alternative implementations for the `change` pointcut: using regular expressions (e.g. `Figure+.set*(..)`), using enumerations (e.g. `Point.setX(..)||Point.setY(..)`), using annotations explicitly applied to the base code, and using annotations as proposed by the solution described in this paper.

Table 5 describes each of the change scenarios. The first scenario (Ch1) prescribes adding a new `color` field to the class `Point` and a corresponding setter method. Pointcut definitions based on regular expressions require that this method starts with `set`. Although this represents a naming convention, it is usually satisfied by setter methods. For this reason, the solution based on regular expressions was classified as robust (+). The solution based on enumerations is not robust (-), because the declared enumeration must be updated with the new setter method. Similarly, the solution based on scattered annotations is not robust, since it requires (but not enforces) developers to annotate the new setter methods. Finally, the Annotator solution was classified as robust, since it will detect that a new method has been inserted into the class and it will ask developers whether this method affects the display or not.

	Change	RE	Enum	Ann	Annotator
Ch1	(Class definition change) Inserting a new `color` field to the class `Point` and a correspondent setter method.	+	-	-	+
Ch2	(Class definition change) Inserting a new `date` field to the class `Point` and a correspondent setter method.	-	+	+	+
Ch3	(Class definition change) Renaming method `setX` from class `Point` to `changeX`.	-	-	+	+
Ch4	(Class hierarchy change) Inserting a new class into the `Figure` hierarchy (such as `Circle`).	+/-	-	-	+
Ch5	(Class hierarchy change) Renaming interface `Figure` to `FigureElements`.	-	-	+	-
Ch6	(Object graph change) Use an object of type `Pair` to store the coordinates of a `Point` and a correspondent setter method.	+/-	-	-	+/-
Ch7	(Control flow change) Inserting a new `enable` field to the class `Point` to control if an object should be exhibited on the display or not.	-	-	-	-

Table 1. Robustness of pointcuts based on regular expression (RE), enumerations (Enum), Java annotations (Ann) and annotations superimposed by the annotator tool, considering seven possible changes to the Figure Editor System.

In the second scenario (Ch2), a field that does not modify the display state is inserted in the class `Point`. In this case, the solution based on regular expressions is not robust, since it will capture the execution of the setter method of this field (assuming as usual that its name starts with `set`). The solution based on enumerations is robust, because the declared enumeration does not need to be updated in this case. Implementations based on explicit annotations and on the annotator tool are also robust.

In the third scenario (Ch3), method `setX` from class `Point` is renamed to `changeX`. This generates an accidental join point miss, supposing pointcut descriptors based on regular expressions and enumerations. On the other hand, it does not impact the robustness of annotation-based pointcuts, because the already defined annotations remain valid after the change. It also does not have impact on pointcuts based on annotations applied by the annotator tool, since developers will inform that the execution of the renamed method continues to affect the state of the display.

In the fourth change scenario (Ch4), a new type (`Circle`) is inserted into the `Figure` hierarchy. In such scenario, regular expressions are partially robust (+/-), because they require that setter methods of visual fields must start with `set` and that non-visual setter methods do not start with this prefix. The solution based on enumerations is not robust, because the declared enumeration must be updated with the methods of the new class. Conventional annotation-based pointcuts are also not robust, because they do not enforce developers to annotate the methods of the new class. Finally, the annotator-based solution is robust, because it prompts developers about the nature of each method of the new class, regarding their display state characteristics.

In scenario Ch5, interface `Figure` is renamed to `FigureElements`. This change does not have impact on annotation-based pointcuts (because the existent annotations remain valid). On the other hand, it breaks implementations based on regular expressions, because such expressions rely on the `Figure` type. Also, the solution based on enumerations is not robust, because the declared enumeration must be updated with the new type name. Change Ch5 also breaks pointcuts defined with the help of the annotator tool, because the scope of `@DisplayStateChange` is declared using the `Figure` type (Section 3).

In scenario Ch6, a field of type `Pair` is used to store the coordinates of a `Point`, instead of using two `int` fields (`x` and `y`). Regarding this scenario, RE-based pointcuts are partially robust, because they will capture the new setter method of the `Point` class (i.e. the `setCoordinates` method that replaces the previous `setX` and `setY` methods). However, it will not capture anymore calls to methods of the class `Pair`, such as `setX` and `setY`. The same happens with pointcuts defined with the help of the annotator tool. The implementation based on enumerations is not robust, because the declared enumeration must be updated with the methods of the new `Pair` class. Conventional annotation-based pointcuts are also not robust, because they require (but not enforce) developers to annotate the setter methods of the new type.

In scenario Ch7, a new boolean field controls if a fig-

ure is enabled or not. Only enabled figures are drawn on the display. Thus, this scenario requires modification in the `change` pointcut signature (in order to expose the target of the call) and also in the `Figure` interface (in order to define a new `isEnabled` method). As a result, neither of the four evaluated implementations of `change` are robust to Ch7.

Table 2 summarizes the robustness of the evaluated implementations of the `change` pointcut, regarding the previous change scenarios. The implementation based on enumeration has presented the lowest degree of robustness. In fact, it was robust to only one of the seven considered scenarios. On the other hand, the solution based on the proposed annotator tool was robust to four changes and partially robust to one change.

Pointcut	+	+/-	-
Regular Expressions	1	2	4
Enumerations	1	0	6
Annotations	3	0	4
Annotator	4	1	2

Table 2. Summary of the results for the seven evaluated change scenarios

6 Discussion

It is well-known that annotations decouple pointcut descriptors from the abstract syntax tree of the base program, which is a key property to handle several problems inherent to the evolution of aspect-oriented systems. However, annotations as currently used in AspectJ also present two important problems: they usually lead to crosscutting code and they are fragile. For example, the following pointcut is decoupled from the abstract syntax of the base program, but it also requires developers to annotate correctly all methods that change the state of the display:

```
pointcut change():
      execution(void @DisplayStateChange *.*(..))
```

The solution proposed in this paper contributes exactly to reduce the fragility and to eliminate the crosscutting behavior inherent to the use of annotations when defining pointcut descriptors. In order to reduce fragility, the solution relies on a simple annotation declaration language to remind developers about program elements subjected to annotations. In order to avoid annotation scattering and tangling, the solution generates annotator aspects that superimpose annotations to the base program in a non-invasive way. Annotator aspects are generated (or updated) whenever developers accept the annotation remind provided by the tool and agree that a candidate program element should be annotated.

The solution is particularly recommended when one of the following two situations happen. First, when it is possible to correlate annotations with structural (and robust) properties that characterize their target elements. For example, in JAccounting only methods that call `Session.save` are candidates to transaction handling. When this first situation happens, the preliminary selection presented by the tool is usually fairly precise (thus avoiding false positives) and also robust to evolutions in the base program (thus avoiding false negatives). Second, we also recommend using the solution when it is possible to restrict the scope of an annotation, as is the case of the Figure Editor and HealthWatcher systems. For example, in HealthWatcher persistent classes always appear as parameter in methods of a single class of the system (the facade class). When this second situation happens, developers are not constantly interrupted by questions asking whether a matched target element is really a valid join point in the system.

The solution also presumes that stable names are used to define the target and scope of annotations. For instance, in JAccounting the method `Session.save` is one of the most important methods provided by the underlying persistence framework. Thus, changing the name of this method is very unlikely. The same happens with the name of the facade class in HealthWatcher. When stable names are not employed, they can compromise the declaration of annotations, as demonstrated by chance scenario Ch5 in the case study of Section 5.

In certain way, our solution is in line with recent trends in software engineering that recognize that current programming languages and tools are inflexible to handle most real-world abstractions (since such abstractions are usually transient, partial, evolving etc) [8]. Thus, instead of designing a new pointcut language, with sophisticated, formal and context-specific abstractions, we decided to follow an approach that recognizes the limitation of formal languages to handle complex abstractions such as pointcuts and join points. Moreover, the proposed solution is lightweighted and flexible, mainly because its semantics depends on feedback given by developers. Nevertheless, we do not claim that our solution solves all the possible instances of the fragile pointcut problem. As described earlier, its effectiveness depends basically on the capability to define the target and scope of annotations in a precise way.

7 Related Work

Annotation-based pointcuts: Explicit annotations are often mentioned as an alternative to design more robust pointcuts. Kiczales and Mezini recommend using annotations when: (i) it is difficult to write a stable regular expression or enumeration-based pointcut; (ii) the name of the annotation is unlikely to change; (iii) the annotation denotes a well-

defined semantic property (and not properties that are only true in some configurations of the system) [10]. Eaddy and Aho propose using annotations at the statement level for exposing join points needed by heterogeneous concerns and for enabling fine-grained advising [3]. However, their proposal can lead to a widespread use of annotation and thus can increase the code scattering and tangling phenomenon usually associated to Java annotations. Havinga et al. have described an expressive pointcut language that supports the superimposition of annotations on selected program elements (in order to avoid annotation scattering and tangling) [6]. Different from the introduction of annotations in AspectJ, their language supports derivation of annotations, i.e. it is possible to introduce an annotation to a program element if another element has a certain annotation. However, their work does not directly tackle the fragile pointcut problem (e.g. forgetting to superimpose an annotation can lead to join point misses).

Logic-based pointcut languages: A common approach to cope with the fragile pointcut problem devises the definition of more expressive pointcut languages. For example, CARMA is a Prolog-like language in which predicates are provided to capture both static and dynamic join point properties [5]. Particularly, CARMA includes predicates to reason about message reception, message send and lexical properties of Smalltalk programs. Alpha is another logic-based pointcut language that supports reasoning about different models of program semantics, including execution traces and heap state [13]. Clearly, expressive pointcut languages contribute to the definition of more robust pointcuts, including for example pointcuts that express structural and behavioral properties of the base program (instead of just capturing join points by source code syntax). On the other hand, expressive pointcut languages usually lead to complex pointcut descriptors, that are difficult to understand and to implement efficiently. More important, such languages do not completely solve the fragile pointcut problem, since pointcuts continue to depend on base program elements [7].

Model-based pointcut languages: Such languages propose that pointcuts should be defined in terms of conceptual models of the base program, instead of directly accessing the program abstract syntax tree. The rationale behind such approaches is that models are more robust to evolution, because they represent only abstract and consolidated concepts about the program domain. Kellens et. al advocate the use of intensional views to capture model-based concepts of interest when defining pointcut descriptors [7]. Moreover, a set of constraints on and between views are proposed to synchronize model abstractions with the base program. Motorola Weavr is a tool that supports weaving at the level of UML models represented by statecharts that include action semantics [2]. As usual in model-based approaches, the tool proposes that pointcut designators should be expressed in terms of stable model elements rather than implementation elements.

Model-based pointcuts share many similarities with the use of annotations to classify source-code elements according to semantic (or model-based) properties, as proposed in this paper. However, we have deliberately left to developers the final decision about keeping annotations synchronized with the base code, instead of requiring developers to express synchronization constraints in a formal language.

Crosscutting interfaces (XPIs): XPIs are explicit, abstract interfaces that decouple aspects from details of advised code [4, 16]. The idea is to support information hiding and parallel development in aspect-oriented systems. XPIs define contracts (or design rules) that base code developers must observe. On the other hand, aspect developers must rely on the syntactic part of XPIs to implement advices that do not directly reference source code elements. According to Sullivan et. al, XPIs support feature obliviousness, in the sense that classes can remain oblivious of possible aspects. On the other hand, classes must be prepared to facilitate the definition of pointcuts. Pointcuts based on XPIs are also more robust, since the base code must adhere to the defined design rules even after changes. However, in its current stage, design rules enforced by XPIs are implemented in AspectJ, which probably is not the best language in terms of expressiveness for design rule definition and checking.

The solution proposed in this paper differs from XPIs in the sense that it does not try to define strict design rules that the base code must conform to. On the other hand, it struggles to find indications that characterize a crosscutting concern even after several changes are applied to base code. As example, we can mention calling `Session.save` in the implementation of transaction handling in the JAccounting system and the existence of methods in the HealthWatcher facade that expose business (and persistent) objects to the different clients of the system.

Confirmed join points: In the confirmed join points approach, class owners need to confirm that matched join points are acceptable [12]. For this purpose, they must change the class code in order to insert confirmation statements, which explicitly name the pointcuts that are allowed to match the join point shadows provided by the class. Confirmed join points is similar to the annotator approach, because both delegate to class owners the final decision about whether a pointcut can or not match join points. On the other hand, our proposal is non-invasive, in the sense that it does not require developers to modify the object-oriented code by inserting `confirm` statements.

8 Conclusions

The fragile pointcut problem is a serious limitation for the widespread use of aspects in large-scale software systems. In this paper, we leverage previous solutions that propose the use of annotations to classify base code elements according to their semantic properties. In our proposal, annotations are superimposed to the base program by annotator aspects, in order to turn annotations in non-scattered and non-invasive elements. Moreover, annotations are defined in a declarative language. This language does not have the purpose to strictly define the elements of the program that should be annotated. Instead, its purpose is to express code patterns that indicate that a given program element may be annotated. The benefit of this approach is the fact that such patterns are usually much less affected by system evolutions. Thus, it is less probable that evolutions require changes in the proposed annotation declaration files. On the other hand, our approach requires developers support in order to eliminate false positives.

The current version of the annotation definition language was able to successfully capture potential annotation targets in two medium-sized AOP systems and in the classical Figure Editor application, with a reduced number of false positives. Moreover, we have also demonstrated that the proposed solution was more robust to changes in the Figure Editor System than traditional pointcuts based on regular expressions, enumerations, and invasive annotations.

We have plans to investigate the use of the proposed solution in other pointcuts of the systems considered in the paper and also in other AOP systems. The objective is twofold: to determine new patterns that can be useful to define the target and the scope of annotations and also to better establish the limitations and the type of crosscutting concerns where using the proposed solution is not recommended.

Acknowledgment: This research was supported by a grant from FAPEMIG.

References

[1] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Transactions Software Engineering*, 32(9):698–717, 2006.

[2] T. Cottenier, A. van den Berg, and T. Elrad. Joinpoint inference from behavioral specification to implementation. In *21st European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *Lecture Notes in Computer Science*, pages 476–500. Springer, 2007.

[3] M. Eaddy and A. V. Aho. Statement annotations for fine-grained advising. In *ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, pages 89–99, 2006.

[4] W. G. Griswold, K. J. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60, 2006.

[5] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 60–69, 2003.

[6] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit. Detecting and resolving ambiguities caused by inter-dependent introductions. In *5th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 214–225, 2006.

[7] A. Kellens, K. Mens, J. Brichau, and K. Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In *20th European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *Lecture Notes in Computer Science*, pages 501–525. Springer Verlag, 2006.

[8] G. Kiczales. Effectiveness sans formality. In *Keynote talk at OOPSLA*, 2007.

[9] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *27th International Conference on Software Engineering (ICSE)*, pages 49–58, 2005.

[10] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *19th European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 195–213. Springer, 2005.

[11] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *12th European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382. Springer, 1998.

[12] H. Ossher. Confirmed join points. In *AOSD Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT)*, pages 1–6, 2006.

[13] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *19th European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 214–240. Springer-Verlag, 2005.

[14] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *17th ACM Conference on Object-Oriented programming systems, languages, and applications*, pages 174–190. ACM Press, 2002.

[15] M. Störzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *21st IEEE International Conference on Software Maintenance (ICSM)*, pages 653–656, 2005.

[16] K. J. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In *13th International Symposium on Foundations of Software Engineering (FSE)*, pages 166–175, 2005.

[17] T. Tourwé, J. Brichau, and K. Gybels. On the existence of the AOSD-evolution paradox. In *AOSD Workshop on Software-Engineering Properties of Languages*, 2003.