Tackling Pointcut Fragility with Dynamic Annotations

Carlos Noguera
Software Languages Lab
Vrije Universiteit Brussel
B-1050 Brussels, Belgium
cnoguera@vub.ac.be

Andy Kellens
Software Languages Lab
Vrije Universiteit Brussel
B-1050 Brussels, Belgium
akellens@vub.ac.be

Dirk Deridder
Software Languages Lab
Vrije Universiteit Brussel
B-1050 Brussels, Belgium
dderidde@vub.ac.be

Theo D'Hondt
Software Languages Lab
Vrije Universiteit Brussel
B-1050 Brussels, Belgium
tjdhondt@vub.ac.be

ABSTRACT

Within the aspect-oriented software development community, the use of annotation-based pointcuts has been proposed as a means to alleviate the fragile pointcut problem. Expressing pointcuts in terms of source-code annotations instead of the structure of the source code, decouples them from the source code of the base system and makes them more robust with respect to evolution. In this paper we demonstrate that, while annotations are suitable to capture static domain knowledge that can be leveraged by pointcut expressions, these annotations are ill-suited to capture dynamic domain knowledge. Consequently, pointcuts that rely on such dynamic knowledge still need to be defined in terms of actual source-code entities, thereby rendering them fragile again. As a means to alleviate this problem we propose Dynamic Annotations, an extension to Java annotations where the dynamic conditions under which the annotation is valid can be embedded in the annotation itself. By expressing pointcuts in terms of such dynamic annotations, these pointcuts are effectively decoupled from the structure of the base program, and become less fragile with respect to evolution.

Keywords

1. INTRODUCTION

In recent years, the use of metadata facilities has gained momentum within the software engineering community. Languages such as C# and Java allow developers to attach additional information to source-code entities by means of annotations. Such annotations have been leveraged by frameworks (for example Spring [17] and Hibernate [2]) to configure, document and influence the behavior of the annotated applications.

RAM-SE 2010

Within the aspect-oriented software development community, annotations have been proposed as a way to alleviate the fragile pointcut problem [8, 12]. This problem states that upon evolution of an aspect-oriented program, seemingly safe changes to the base code of that system can have an unexpected impact on the aspects that advise this base code. The cause of this problem is the fact that, in order to select the correct set of join points, pointcut definitions are often tightly coupled to the structure of the base code. To capture the correct intent of a pointcut definition, developers rely on the presence of (implicit) structural and naming conventions. Consequently, if these conventions are violated in the base code, the pointcut will capture an incorrect set of join points.

Expressing pointcuts in terms of source code annotations has been proposed [6, 10] as a means to tackle the fragile pointcut problem. By annotating source code structures (such as classes, fields, methods) developers can expose domain concepts in the source code of a system. This allows pointcuts to be defined in terms of those domain concepts, thereby decoupling the pointcut definitions from the code's structure. In other words, annotations introduce an indirection layer between the aspects and the advised base code, making the pointcut definitions less fragile with respect to changes of the base code.

However, pointcuts can also contain dynamic conditions (i.e. using the `if` pointcut designator), which are matched during the execution of the program. Since these dynamic conditions are expressed in terms of the base language, they inherently refer to base program source-code elements (variable names, method signatures, ...). For example, a pointcut that needs to capture all adults will do so by an expression similar to `p.age > 18`. Although this is a dynamic condition it is tightly coupled to the structure of the source code (the `age` field in the class `Person`). As a result, dynamic conditions are as tightly coupled to the source code's structure as static pointcut designators, making them equally fragile. Since annotations are associated with static source code entities, they are well-suited for capturing the static parts of pointcuts, but provide no help in avoiding fragility of dynamic conditions.

This paper introduces Dynamic Annotations. Our approach augments annotations with *dynamic* conditions that govern their activation in contrast to regular annotations

```
public class Bank {
 Collection<Account> accounts = new ArrayList<Account>();

 public Account openAccount(){
  Account a = new Account();
  accounts.add(a);
  return a;}

 @Financial
 public void withdraw(double amount, Account a){
  a.balance -= amount;}

 @Financial
 public void transfer(double amount, Account from,
                                     Account to){
  from.balance -= amount;
  to.balance += amount;}

 @Financial
 public void deposit(int amount,Account a){
  a.balance += amount;}
}
```

Figure 1: Motivating example – Bank system

that are *statically* associated with source code elements. This mechanism allows developers to use annotations to represent both static and dynamic domain knowledge. A developer can expose the concept of adults by annotating the `Person` class with the dynamic annotation `@Adult(target.age > 18)`. By basing pointcut definitions on DYNAMIC ANNOTATIONS, we can further reduce pointcut fragility, decoupling dynamic conditions in pointcuts from the structure of the base code.

The rest of the paper is structured as follows: in the next section we will present and discuss an example that highlights the shortcomings of current aspect and annotation technologies when dealing with crosscutting concerns that depend on dynamic conditions in terms of the evolvability of the system. In Section 3 we will present our approach and its current implementation. The example is revisited in Section 3.2 where DYNAMIC ANNOTATIONS is used. Section 4 presents existing work which relates to our approach, and Section 5 summarizes the paper and concludes.

2. MOTIVATING EXAMPLE

As a motivating example, consider the implementation of a simple bank system, part of which is shown in Figure 1. This banking system keeps track of a number of accounts, and offers operations to open accounts, deposit and withdraw money from an account, and transfer funds between two accounts. Methods that manipulate the amount of money in an account (`withdraw`, `transfer`, `deposit`) are annotated in the source code of the system with the annotation `@Financial`. These annotations expose part of the domain knowledge regarding the annotated methods, namely that they implement a financial operation.

Now suppose that we want to extend this system to ask approval of certain financial operations, that — due to the amount of money involved — are considered risky. Because of legislation regarding money laundering we consider withdrawals over 500 euros and transfers over 1500 as risky. Additionally, any transfers below 1 cent are also considered suspicious, as they might indicate a salami attack where very small amounts of money are siphoned from a number of accounts. Since the approval functionality cross-cuts the implementation of various financial operations, implementing it as an aspect is natural choice. We will begin by implementing this aspect using traditional techniques, and then elaborate on the issues raised by such an implementation. Afterwards we will introduce DYNAMIC ANNOTATIONS that provide a more elegant solution to the problem.

2.1 AspectJ Implementation

Figure 2 shows the aspect that implements the approval of risky financial operations. This aspect is implemented by three pointcuts to select risky financial operations and an around advice that will ask for their approval. The first pointcut `financialOp` captures all the financial operations in the system. It is defined in terms of the `@Financial` annotation and selects execution join points corresponding to methods that are annotated with this annotation, and that take as a first argument a `double` which will be bound to the variable `amount` of the pointcut. While this pointcut allows to capture all financial operations in the system, it needs to be further refined to capture *risky* financial operations. Since the condition for being a risky financial operation depends on the type of operation, i.e. a withdrawal or a transfer, we cannot describe them in a single pointcut. Rather, we specify two pointcuts `limitwithdraws` and `limittransfer` that use the `financialOp` pointcut. The `limitwithdraws` pointcut captures all executions of the `withdraw` method where the withdrawn amount is larger than 500 euro; `limittransfer` captures the executions of `transfer` where the amount is either larger than 1500 euro or lower than 1 cent. The actual approval of risky operations is implemented by an `around` advice that requires a user to confirm the operation before allowing the call to `proceed()` to happen.

2.2 Discussion

The implementation of the `ApprovalOfRiskyOps` aspect is fragile. Although it relies on annotations to select financial operations, it is still coupled to the base code structure for the selection of risky amounts. The first pointcut (`financialOp`) is defined in terms of the `@Financial` annotation and, because of this, it is loosely coupled with the source code. Given that knowing which methods implement financial operations does not depend on runtime information, it can be easily exposed by an inherently static source code annotation. If new methods that implement financial operations are added to the base code, the pointcut will correctly capture them, provided that they carry the correct annotation. This is possible because the definition what constitutes a *financial operation* (i.e., the `withdraw`, `deposit` and `transfer` methods) is transferred from the pointcut expression to the base code, thus decoupling aspect and base code.

Decoupling the definition of what constitutes a risky operation in the same manner is, however not possible. A financial operation is considered risky based on the type and the dynamic context of the operation. A withdrawal is risky if the amount is larger than 500 euro; transfers are risky if the amount is larger than 1500 euro or smaller than 1 cent. These dynamic conditions cannot be easily captured by means of annotations. Therefore, different pointcuts (`limitwithdraws` and `limittransfer`) are necessary to identify and discern between the various types of risky operations. As can be seen in Figure 2, these pointcuts refer directly the particular base source-code entities and are

```
public aspect ApprovalOfRiskyOps {
 pointcut financialOp(double amount) : execution(@Financial * *.*(double,..)) && args(amount,..);
 pointcut limitwithdraws(double amount) : financialOp(amount) && execution(* *.withdraw(..)) && if(amount > 500);
 pointcut limittransfer(double amount) : financialOp(amount) && execution(* *.transfer(..))
             && if(amount > 1500 || amount < 0.01);

 void around(double amount) : limitwithdraws(amount) || limittransfer(amount) {
  // ask approval
  if(answer.equalsIgnoreCase("y"))
    proceed(amount);}
}
```

Figure 2: AspectJ implementation of approval of risky financial operations

therefore tightly coupled with these entities, rendering the pointcuts brittle with respect to evolution and negating the benefits of the use of `@Financial` in the `financialOp` pointcut. For example, the addition of new risky financial operations in the base code requires one of the pointcut definitions to be extended (in case the dynamic conditions of the new operation are identical to those of an existing one) or a new pointcut for this operation to be defined. Notice that the introduction of an annotation that exposes potentially risky operations does not solve this problem, since the actual conditions that determine whether an operation is risky differ for each of the various kinds of operations, something which cannot be captured by the annotations.

3. DYNAMIC ANNOTATIONS

The above example demonstrates the limited capabilities of static Java annotations to expose domain knowledge that is inherently dynamic. To overcome this limitation, we introduce Dynamic Annotations. Dynamic annotations extend the concept of regular Java annotations with a means to incorporate the dynamic conditions that govern their activation. A dynamic annotation is defined as a regular Java annotation type with a special member, denoted by a marker annotation `@ObjectExpression`. An annotation's object expression evaluates to `true` when the annotation should be active. Context information is passed to the expression by means of a `target` metavariable. This variable is bound to the object instance on which the annotation is placed. Note that, because of this, Dynamic Annotations are not allowed on methods.

To illustrate the concept of dynamic annotations, consider an annotation `@Adult` defined in Figure 3. The `Adult` annotation type contains a single member, `value` that serves as the annotation's object expression. By default, all uses of the annotation are active, as expressed by the default object expression `true`. The `@Adult` annotation is used to expose that particular instances of the `Person` class are considered adults, i.e., those whose age field is greater than 18. To express this dynamic condition, the `Person` class carries the dynamic annotation `@Adult(target.age > 18)`. The use of a dynamic annotation to express the domain-concept of a person being an adult serves as a data-driven interface to aspects, who are now shielded from changes on the internal structure of the `Person` class.

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Adult {
        @ObjectExpression
        String value() default "true";}

@Adult("target.age > 18")
public class Person {
        String name;
        private int age;}
```

Figure 3: `Adult` Dynamic Annotation and its use on the `Person` class

3.1 Implementation

We have implemented a first prototype of Dynamic Annotations[1] as a library that uses the Java reflection API and Groovy [11] as an object expression language. The Dynamic Annotations runtime defines an `AnnotationManager` class with a single static method, `isActive(Object, Annotation)` to check whether a dynamic annotation is active on a given object. This method uses the reflection API to find out the object expression defined in the annotation. Then, it uses Groovy's shell to evaluate the object expression in the context of the queried object. We use Groovy because of its symbiosis with Java, and the ability to evaluate groovy expressions dynamically. In principle, any language that allows this is a viable choice to describe object expressions. Groovy has the advantage of having a syntax similar to Java's, which enhances the understandability of dynamic annotations embedded in the source code.

Aspects that want to take Dynamic Annotations into consideration in their pointcuts can use the `AnnotationManager` to query for the state of an annotation in a given object. This can be achieved by capturing the dynamic annotation and the object that is annotated using existing pointcut descriptors, and querying the `AnnotationManager` means of the `if()` PCD. In the case of instances of the class `Person` and the `@Adult` annotation, a possible pointcut would look like this:

```
pointcut foo(Person p, Adult a) :
  execution(* *(..)) && target(p) && @annotation(a)
  && if(AnnotationManager.isActive(p,a))
```

Although this way of selecting joinpoints based on dynamic annotations has the advantage of using existing AspectJ semantics, PCDs specific to dynamic annotations would enhance the readability of aspects that use them. Introducing such PCDs, even as syntactic sugar, remains a task for future work.

3.2 Example Revisited

To illustrate the applicability of dynamic annotations, we update the implementation of the banking system to use our

[1]that can be downloaded from `http://soft.vub.ac.be/soft/research/sustainablecode/dynamicannotations`.

```
@Financial
public void withdraw(@Risky("target>500") double amount,
        Account a){
 a.balance -= amount;}

@Financial
public void transfer(
        @Risky("0.01<target || target>1500") double amount,
        Account from, Account to){
 from.balance -= amount;
 to.balance += amount;}
```

Figure 4: Updated bank system code (fragment)

```
public aspect ApprovalOfRiskyOps {
 pointcut riskyOperations(double amount, Risky risky) :
        execution(@Financial * *(..))
        && args(amount,..)
        && @annotation(risky)
        && if(AnnotationManager.isActive(amount,risky)) ;

 void around(double amount, Risky risky) :
    riskyOperations(amount, risky) {

   // ask approval
   if(answer.equalsIgnoreCase("y"))
   proceed(amount,risky);}
```

Figure 5: Updated aspect to implement the approval of risky financial operations (fragment)

approach. Figure 4 shows the updated source code of the base code of the system, where the domain concept of risky financial operations is now exposed using the dynamic annotation `@Risky`. For the `withdraw` and `transfer` methods, the first argument (representing the `amount` of the financial operation) is annotated with the `@Risky` annotation[2]. For both methods, this dynamic annotation contains a boolean expression that exposes under which conditions the operation can be considered risky. For example, the execution of the `withdraw` method is considered risky if the value of the amount (`target`) is larger than 500.

The updated version of the `ApprovalOfRiskyOps` aspect can be found in Figure 5. Rather than requiring three different pointcuts as was the case with the standard AspectJ solution, this aspect consists of a single pointcut named `riskyOperations`. This pointcut selects all execution join points of methods that are annotated with the `@Financial` annotation for which the `@Risky` annotation (dynamically) holds. The actual verification of whether the `@Risky` annotation is active or not in the context of a given join point is performed by the last line of the pointcut definition. The pointcut consults the `AnnotationManager` class which is part of the implementation of Dynamic Annotations. Given an instance of the annotation (`risky`) and the concrete value of the annotated object (`amount`), this `AnnotationManager` evaluates the dynamic conditions corresponding to the annotation with respect to the annotated object, and returns whether or not the annotation applies. Note that the `amount` and `risky` objects are exposed in the pointcut not because they are necessary to the advice execution, but because AspectJ requires all values bound in the pointcut to be exposed.

[2]While Java allows arguments to be annotated, AspectJ does not allow such argument annotations to be captured in pointcuts (AspectJ Bug #259416). As a consequence, in our concrete implementation we annotated the `transfer` and `withdraw` methods with the `@Risky` annotation. This is however a limitation of AspectJ and not of our approach.

3.3 Discussion

The use of dynamic annotations to express the concept of financial operations on risky values brings benefits to both the aspect that implements their approval and the base program.

Compared to the traditional AspectJ implementation of the `ApprovalOfRiskyOps` aspect, the version that uses dynamic annotations is less fragile with respect to evolution of the base code. Rather than referring to concrete base source-code entities, the pointcut definition leverages the domain concepts that were exposed in the base system by the static `@Financial` and the dynamic `@Risky` annotations. By relying on these domain concepts instead of on how the base program is structured, the `ApprovalOfRiskyOps` aspects will capture the correct set of join points upon evolution of the system, under the premise that the base code developer correctly annotates all (risky) financial operations. While the incorrect use of annotations can be a source of pointcut fragility, tools have been proposed such as [9, 18] that offer support to co-evolve source code and annotations.

At the level of the base program, dynamic annotations extend the expressiveness of standard Java annotations. In the example, this translates into the explicit expression of what constitutes a risky operation. The semantics of such operations is further refined by the dynamic annotations into operations that deal with risky values. Since in this version of the Banking system, amounts are represented as primitive types, it is not possible to delegate the responsibility of deciding whether an amount is risky to the object that represents it. The addition of the `@Risky` annotation also serves as explicit documentation of the concept. As the example demonstrates, the use of dynamic annotations offers base code developers an elegant means to expose domain concepts that are dynamic in nature. By allowing developers to parameterize annotations with the conditions that define whether or not the annotation is active at runtime, annotations become context-sensitive and can express concepts that not only depend on the annotation being present in the source code, but also on the runtime context of the system.

Using dynamic annotations also enhance the evolvability of the system. Changes to what constitutes a risky value, or to the way in which amounts are represented, are local to the base code.

4. RELATED WORK

In this section we compare Dynamic Annotations to existing work, both on the annotation and the aspect-oriented community. Annotations in Java have been used to as a means to express a pluggable type system [14]. In this sense, Dynamic Annotations are close to pluggable dependent types, since the annotation of an object depends on the value of the object. The Checker framework [15], built using type annotations [4], provides the notion of dependent annotation types, by conditioning the annotation type of an object to the presence of another annotation type. The difference between our approach and dependent types in the Checker framework lays in that Dynamic Annotations are not used to check the validity of a program, but they are used to ex-

pose (dynamic) domain concepts.

Decorating classes and methods with runtime expressions is usually the approach taken in Design-by-contract frameworks such as JML [16], J-LO [3] and Contract4J [13]. In the two former frameworks, annotations with boolean expressions are added to methods and classes in order to specify pre/post conditions and class invariants. DYNAMIC ANNOTATIONS can be used to specify such contracts defining a dynamic annotation that represents the concept of a valid state, and an aspect that throws an exception whenever a method that is not in a valid state (i.e., a method whose dynamic annotation is not active) is executed.

The use of DYNAMIC ANNOTATIONS to deal with fragile pointcuts does so at the expense of obliviousness of the base code. Approaches like Open Modules [1] and Crosscutting interfaces (XPI) [5] also forgo obliviousness by specifying rules on how should aspects interact with the base code to tackle the problem of aspect interaction. Both these approaches allow base-code developers to express an aspect interface on which aspects and base code agree in order to enhance the modularization of the composed system. To our knowledge, however, neither XPI or Open Modules allow the explicit definition of dynamic conditions as part of the aspect interface.

In terms of pointcut fragility, annotations [6, 10] and explicit joinpoints [7] embed references to crosscutting concerns in the base code itself. In the case of annotations this is achieved by exposing domain concepts. Explicit joinpoints directly reference the aspects that should intervene at a particular location in the base code. These explicit joinpoints however do not allow for capturing dynamic context.

5. SUMMARY

In this paper we present DYNAMIC ANNOTATIONS, an extension to the Java metadata facility. Dynamic annotations allow the developer to attach metadata to source code entities, and to condition the activation of the metadata to the runtime context. Dynamic annotations address the issue of pointcut fragility in AspectJ by allowing developers to annotate base-code entities with domain concepts that depend on runtime values. These annotations can then be leveraged by aspects, avoiding `if` point cut designators that refer to the structure of the base code, and replacing them with references to the dynamic annotations instead. We illustrate the kind of situations in which DYNAMIC ANNOTATIONS are useful by means of an example that implements the approval of risky financial operations on a banking system. The modularization of these kind of operations on an aspect results in a fragile pointcut if dynamic annotations are not used.

6. ACKNOWLEDGMENTS

Andy Kellens is funded by a research mandate provided by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen). This research is supported by the IAP Programme of the Belgian State.

7. REFERENCES

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 144–168. Springer, 2005.

[2] C. Bauer and G. King. *Java Persistence with Hibernate*. Manning Publications, 2006. ISBN 978-1932394887.

[3] E. Bodden. A lightweight LTL runtime verification tool for Java. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28,*, pages 306–307, 2004. Student Research Competition.

[4] M. D. Ernst. Type Annotations specification (JSR 308). `http://types.cs.washington.edu/jsr308/`, September 12, 2008.

[5] W. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Teware, Y. Cai, and Rajan.H. Modular software design with crosscutting interfaces. *IEEE Software, Special Issue on Aspect-Oriented Programming*, 23(1):51–60, January/February 2006.

[6] W. Havinga, I. Nagy, and L. Bergmans. Introduction and derivation of annotations in AOP: Applying expressive pointcut languages to introductions. In *European Interactive Workshop on Aspects in Software (EIWAS)*, 2005.

[7] K. Hoffman and P. Eugster. Bridging java and aspectj through explicit join points. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 63–72, New York, NY, USA, 2007. ACM.

[8] A. Kellens, K. Mens, J. Brichau, and K. Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In *European Conference on Object-Oriented Programming (ECOOP)*, number 4067 in LNCS, pages 501–525, 2006.

[9] A. Kellens, C. Noguera, K. De Schutter, C. De Roover, and T. D'Hondt. Co-evolving annotations and source code through smart annotations. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pages 119–128. IEEE Computer Society Press, 2010.

[10] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *European Conference on Object-Oriented Programming (ECOOP)*, LNCS, pages 195–213. Springer Verlag, 2005.

[11] D. Koenig, A. Glover, P. King, G. Laforge, and J. Skeet. *Groovy in Action*. Manning publications, 2007. ISBN: 1-932394-84-2.

[12] C. Koppen and M. Stoerzer. Pcdiff: Attacking the fragile pointcut problem. In *European Interactive Workshop on Aspects in Software (EIWAS)*, 2004.

[13] I. A. Krizsan. Getting started with contract4j. `http://polyglotprogramming.com/papers/Getting_Started_with_Contract4J.pdf`.

[14] S. Markstrum, D. Marino, M. Esquivel, T. Millstein, C. Andreae, and J. Noble. Javacop: Declarative pluggable types for java. *ACM Trans. Program. Lang. Syst.*, 32(2):1–37, 2010.

[15] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 22–24, 2008.

[16] E. Poll, P. Chalin, D. Cok, J. Kiniry, and G. T.

Leavens. Beyond assertions: Advanced specification and verification with jml and esc/java2. In *In Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures, volume 4111 of LNCS*, pages 342–363. Springer, 2006.

[17] Spring Application Framework. http://www.springframework.org.

[18] E. Tilevich and M. Song. Reusable enterprise metadata with pattern-based structural expressions. In *International Conference on Aspect Oriented Software Development (AOSD)*, 2010.