

Role Annotations and Adaptive Aspect Frameworks

Linda M. Seiter
 John Carroll University
 20700 North Park Boulevard
 University Heights, Ohio, USA
 1-216-397-1948
 lseiter@jcu.edu

ABSTRACT

This paper presents a model for improving the adaptiveness of AOP frameworks. While a metadata pointcut can be an effective tool for encapsulating and consuming the program events related to a crosscutting concern, AOP languages do not provide a succinct mechanism for binding certain types of crosscutting object references used in advice. We present a design pattern for writing adaptive aspects that consume role-based annotations, enabling a framework to be written in a manner that diminishes the occurrence of the fragile pointcut problem.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance and Enhancement – *restructuring, reverse engineering, and reengineering*. D.3.3 [Programming Languages]: Language Constructs and Features – *frameworks, patterns*

1. INTRODUCTION

A framework is typically implemented in AOP through the use of an abstract aspect that contains *template advice*, in which the skeleton of a crosscutting algorithm is defined [3, 11]. The advice sets up the default algorithm to be executed at certain join points within an application. The goal in developing an AOP framework is to allow a crosscutting concern to be applied to a wide variety of applications. However, the method signatures of the target join points in the various applications may not follow a particular pattern. Prior to the introduction of meta-data based pointcuts, a pointcut might require explicit listing of each target method signature. This explicit listing of signatures may result in the fragile pointcut problem, which arises when a pointcut accidentally captures or misses certain join points when the base program evolves [6, 10].

```
@Pointcut("execution( @ConflictTest * *.(..))")
protected void conflictTest() {}
```

Figure 1. Metadata pointcut

Metadata-based pointcuts have been recommended as a way of avoiding the fragile pointcut problem [2,4,5]. If the method

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Workshop LATE '07, March 12-13, 2007, Vancouver, British Columbia, Canada.
 Copyright 2007 ACM 1-59593-655-4/07/03.

signatures of the target join points possess a common annotation, a pointcut can be written to consume the annotation. For example, the @AspectJ-style [14] pointcut in Figure 1 will select the execution of any method whose signature contains the @ConflictTest annotation. A metadata pointcut is adaptive to changes in method signatures, as long as the @ConflictTest annotation remains consistent. Thus, the use of metadata allows a modular, adaptive referencing mechanism for selecting join points, which facilitates evolution.

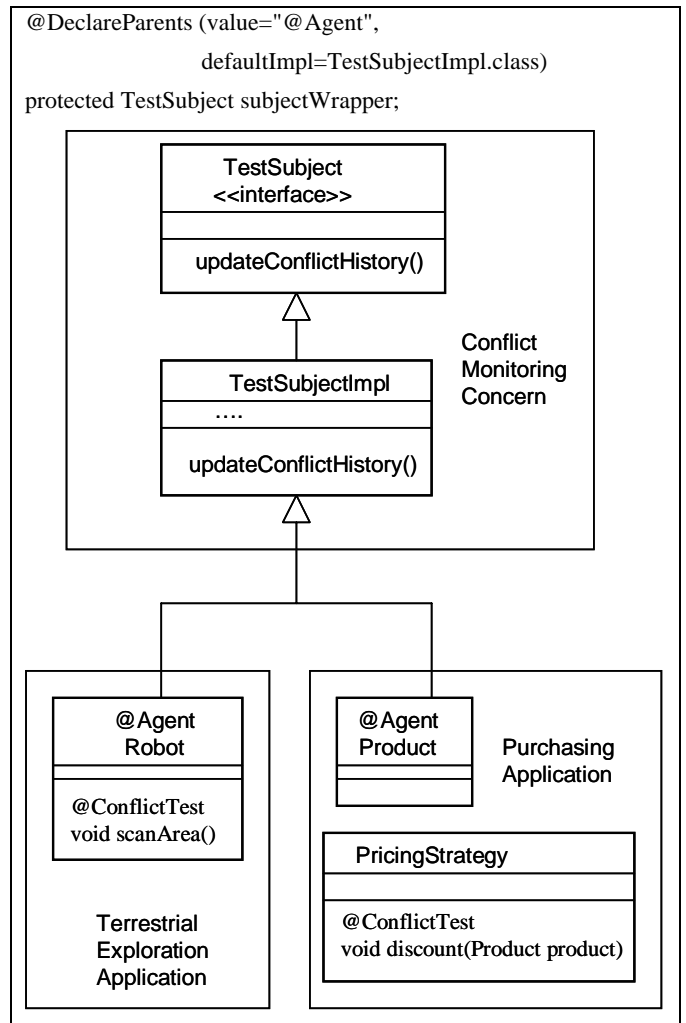


Figure 2. Metadata declaration

The template advice developed in a framework often must view objects not in terms of the core concern's data model, but rather in

terms of a data model that is specific to the crosscutting concern. In addition to implementing the template advice, a framework must also enhance base level classes with additional state and/or interfaces relevant to the crosscutting concern. This is normally achieved through static crosscutting, which produces a *template data model* to be used within the template advice. Static crosscutting may require an explicit 1 to 1 binding of an application class to the wrapper class, unless the set of classes to be adapted follow a naming pattern or shared a common superclass, interface, or annotation. Figure 2 shows an example of an AspectJ advice declaration that extends any class containing the `@Agent` annotation with the `TestSubjectImpl` functionality and `TestSubject` interface. The advice will cause the *Robot* and *Product* classes to inherit from the `TestSubjectImpl` class.

Metadata thus can be used to develop modular and adaptive referencing constructs for dynamic and static crosscutting. However, AOP languages have not yet been able to use metadata to capture certain kinds of crosscutting object references that are found in template advice. We will discuss this problem by presenting a simplified version of an AOP framework developed to optimize existing multi-agent swarm applications [8,9]. The framework introduces a layer of observers that monitor the conflicts that arise during agent collaborations. The results collected during the observation can be used to experiment with different conflict resolution techniques to avoid system thrashing.

```

@Aspect
public abstract class ConflictMonitor {
    //mixin TestSubjectImpl implementation and TestSubject interface
    @DeclareParents (value="@Agent",
                    defaultImpl=TestSubjectImpl.class)
    protected TestSubject subjectWrapper;

    @Pointcut("execution(@ConflictTest * *.(..))")
    protected void conflictTest() {}

    //update agent after conflict assessment
    @After("conflictTest()")
    public void afterConflictTest(JoinPoint jp) {
        TestSubject subject = getTestSubject(jp);
        subject.updateConflictHistory();
    }

    protected abstract TestSubject getTestSubject (JoinPoint jp);
} //aspect ConflictMonitor

```

Figure 3. Abstract Conflict Monitoring Aspect

Figure 3 contains part of an AOP framework for implementing the conflict monitoring and thrashing detection concern. The abstract aspect *ConflictMonitor* defines the template algorithm for manipulating an agent based on its historical record of conflicts with other agents. The pointcut *conflictTest* captures the execution of any method that has been annotated as

@ConflictTest. The template advice serves to update an agent's conflict history based on the result of the test. This information will eventually be used to detect system thrashing and ultimately adjust the agents' conflict resolution strategies. The *ConflictMonitor* aspect requires some object within the scope of a *@ConflictTest* annotated method to play the role of the test subject, i.e. the agent under test. This object is returned by the role binding method *getTestSubject()*.

Since thrashing detection and avoidance is an emergent system-level property and is not part of a swarm agent's core concern, the state needed to maintain the conflict history is part of the crosscutting data model, shown previously in Figure 2. The conflict monitoring framework specifies the crosscutting data model, namely the *TestSubject* interface and corresponding *TestSubjectImpl* class, providing an agent with additional state and methods for maintaining its conflict history. The aspect in Figure 3 uses the *@DeclareParents* advice to perform the static crosscutting to any application type annotated as *@Agent*.

Figure 2 contains classes from two different swarm applications. The *Robot* class belongs to a terrestrial exploration application that uses swarm techniques to implement a dispersion algorithm, attempting to distribute robots evenly over some area of terrain. The *Robot.scanArea()* method checks a robot's local vicinity to see if it has too many neighboring robots. The *DiscountStrategy* and *Product* classes belong to a purchasing application, which uses a swarm of agents attempting to optimize product pricing and inventory management through a set of pricing strategies. The *DiscountStrategy.discount(Product)* method attempts to perform a price adjustment on the product that was passed into the method as a parameter.

Robot.scanArea() and *DiscountStrategy.discount(Product)* both involve some type of conflict detection among agents. Within each method, a particular object is the agent under test, i.e. it is the subject of the conflict test. Table 1 summarizes the mapping of the conflict test subject for the two applications.

Table 1. Test Subject Mapping

Application	Conflict Test Method	Agent Under Test
Robotic	scanArea	Method Target
Purchasing	discount	Method Parameter

Because the two swarm applications reference the test subject at the execution join point using different object references (join point target vs join point parameter argument), it is not possible to write the aspect in Figure 3 to allow the test subject to be passed into the *after* advice as a parameter. Thus, the metadata-based pointcut of Figure 3 is not an effective referencing mechanism when different join points require distinct bindings. In this case, the advice must take the join point as a parameter, and defer the binding of the *subject* reference to concrete aspects that will subclass the abstract *ConflictMonitor* aspect.

The robotic and purchasing applications will require two separate concrete aspects to be developed, each overriding the *getTestSubject* method to bind the subject accordingly. The concrete *DiscountConflictManager* aspect for the purchasing application is shown in Figure 4. A similar aspect would be

required for the robotic application, which is not shown. The *DiscountConflictManager* must bind the test subject as the parameter passed into the *discount* method. In addition to the application-specific subject role binding, each test subject (Product, Robot) must be extended with state and behavior to be manipulated by the template advice from the *ConflictMonitor* aspect. The concrete *DiscountConflictManager* aspect must enable the static crosscutting by adding the *@Agent* annotation to the *Product* class, assuming the annotation was not already present in the application-level class.

The code in Figure 4 demonstrates that metadata-based pointcuts do not prevent the fragile pointcut problem from arising when implementing an AOP framework. While the crosscutting algorithm can be modularized using a metadata pointcut in the abstract aspect *ConflictMonitor* of Figure 3, the binding of the crosscutting *subject* reference requires a signature-based pointcut if all join points do not reference the underlying base object in the same way. The concrete aspects must override the inherited metadata-based *conflictTest* pointcut in order to attach the appropriate *getTestSubject()* implementation for the two different swarm applications. The signature-based pointcut in Figure 4 may lead to maintenance issues when the application evolves.

```
@Aspect
public class DiscountConflictMonitor extends ConflictMonitor {
    //annotate Product as @Agent
    declare @type : Product: @Agent;

    //override conflictTest() pointcut
    @Pointcut("execution(* DiscountStrategy.discount(Product))")
    protected void conflictTest() {}

    //bind test subject to discount method parameter (Product)
    protected TestSubject getTestSubject(JoinPoint jp) {
        return (TestSubject) jp.getArgs()[0];
    }
}
```

Figure 4. DiscountConflictMonitor Aspect

2. ADAPTIVE ROLE REFERENCING

This section presents an approach for facilitating evolution in AOP frameworks. We first reduce the occurrence of the fragile pointcut problem by utilizing role annotations rather than signature-based pointcuts for binding a crosscutting object reference in the template advice. We then eliminate the manual effort required for static crosscutting by automatically generating *@DeclareParents* advice based on the role annotations.

```
public class Robot {
    @ConflictTest(subject="this")
    public boolean scanArea() {...}
}

public class DiscountStrategy {
    @ConflictTest(subject="product")
    public boolean discount(Product product)
    {...}
}
```

Figure 5. Annotating a Crosscutting Object Reference

We propose that the binding of the crosscutting object references used in the template advice be moved directly into the *@ConflictTest* annotation for each application method, as shown in Figure 5. The *ConflictMonitor* aspect can now be rewritten as shown in Figure 6, binding the *subject* reference at the captured join point directly by using a new annotation *@BindRole* which is attached to the *after* advice. The new annotation *@QueryDeclareParents* will adaptively infuse the static crosscutting into any application class that plays the role of the test subject.

```
@Aspect
public class ConflictMonitor {
    //static crosscutting
    @QueryDeclareParents(value="@ConflictTest.subject()",
        defaultImpl=TestSubjectImpl.class)
    private TestSubject subject;

    //dynamic crosscutting
    @Pointcut("execution(@ConflictTest * *.(..)) &&
        @annotation(conflictAnnotation)")
    protected void conflictTest(ConflictTest conflictAnnotation) {}

    //use annotation from join point to bind subject reference
    @BindRole(annotation="conflictAnnotation",
        element="subject", role="TestSubject subject")
    @After("conflictTest(conflictAnnotation)")
    public void afterConflictTest(TestSubject subject) {
        subject.updateConflictHistory();
    }
}
```

Figure 6. Concrete ConflictMonitor Aspect

We utilize annotation processors such as APT[15] and the Java 6 compiler to generate AspectJ code that supports the binding of crosscutting object references as well as the infusion of the static crosscutting. The *@QueryDeclareParents* annotation processor searches the existing workspace for *@ConflictTest* annotations, extracting the *subject* member. The type of the reference is

obtained using static analysis, and boilerplate `@DeclareParents` advice is generated for each such application class. This will alleviate the programmer from having to manually annotate each application-level class whose instances are subjects of a conflict test. Thus, the *Robot* and *Product* classes need not be manually decorated with the `@Agent` annotation; the annotation processor will automatically generate the necessary `@DeclareParents` advice to cause them to be statically crosscut with the `TestSubjectImpl` implementation. The `@BindRole` annotation processor evolves the aspect's template advice to declare and assign a new *subject* local variable to the result of evaluating the expression contained in the `@ConflictTest` annotation's *subject* element. The annotation's *subject* element is dynamically evaluated at the join point using Jexl, the Java Expression Language engine[16]. This in affect allows us to write the advice so that it can expect a bound reference to the appropriate test subject.

3. Discussion

Given the updated ConflictManager aspect of Figure 6, we no longer need two concrete aspects, each containing a fragile signature-based pointcut, to be written for the two separate application methods. The fragile pointcut problem is eliminated since we consume the desired execution join points using the `@ConflictTest` metadata pointcut.

The application-specific role binding needed by the template advice has been moved into the `@ConflictTest` annotations attached to each application method. This may however lead to “the fragile annotation” problem. While the role annotation is more adaptive to changes in the method signature than a pointcut, it may still require maintenance. For example, if the discount method evolves its parameter list, the `@ConflictTest` annotation may need to evolve. Note that the role annotations are written using the application-level object references (this, product) rather than AspectJ pointcut references (getTarget(), getArgs()), which may ease maintenance efforts by taking advantage of source level refactoring tools.

Another possible issue with using the role annotations is deciding whether they violate the obliviousness principle. If the application methods are directly annotated only to support the needs of the consuming aspect, then the annotations are intrusively modifying the application. In the example shown, conflict testing is a core concern of a swarm agent, thus it is reasonable to place the `@ConflictTest` annotation directly on the application method. Conflict monitoring and thrashing avoidance is a crosscutting concern that is implemented as a layer over the agent conflict testing behavior.

Note that while the examples contained in this paper are presented in terms of AspectJ, similar problems arise in other aspect languages. For example, while CeasarJ [12] and ObjectTeams [13] provide far more powerful constructs for statically and dynamically wrapping an object with crosscutting structure than AspectJ, an object reference that must be bound at a join point may result in the same issues as discussed in this paper if different join points require different bindings.

In the remainder of this section we evaluate the proposed technique in terms of its potential impact on aspect mining, concern exploration, aspect extraction, and aspect evolution.

The conflict monitoring framework has been tested with several swarm applications [8,9]. The first application used a swarm of agents to implement an efficient graph coloring algorithm, where each vertex was assigned an agent responsible for resolving color conflicts with adjacent vertices. The original version of the graph coloring application presented a single view of the simulation. AOP was used to extend the simulation with a set of new views that could depict the state of the simulation based on the stability of an agent with its current color choice. Agent stability was determined by the number of iterations without a color conflict. The new views provided numerous insights into how cluster of stable vertices formed and dissolved, which led to strategies for dynamically tweaking the agent color conflict resolutions. The new conflict resolution strategies proved successful in reducing the levels of agent thrashing, which previously had resulted in the frequent dissolution of partial solutions.

A similar model was subsequently added to an existing robotic terrestrial exploration application, using AOP to develop additional views of robot stability based on the presence or absence of overcrowding. The new stability-based views of the robotic swarm again led to insights into how to tweak the robot conflict resolution strategies to avoid thrashing.

After implementing separate sets of aspects for adding the conflict monitoring concern to the two swarm applications, we manually mined the aspect code, and subsequently manually evolved the separate aspect implementations into a conflict monitoring AOP framework. The framework is being applied to additional swarm applications at John Carroll University Swarm laboratory [8]. A retrofit of an existing swarm application to use the conflict monitoring framework presently requires manual human effort, since the existing code must be manually mined to search for and annotate agent conflict detection and resolution methods.

Many AOP frameworks in existence employ signature-based pointcuts to implement static crosscutting and role mapping. It is possible to automatically evolve some of these frameworks to use the role-based annotation design pattern presented in this paper. We are presently developing an Eclipse plugin to mine existing AOP frameworks for occurrences of the signature-based role mapping. The eventual goal is to enhance the plugin to automate the refactoring of such aspects to use role annotations.

4. CONCLUSION

Many techniques have been proposed for lifting a core concern object to its crosscutting concern role [1,3,7,11]. The majority of these however rely on signature-based pointcuts to be developed. The primary contribution of this paper is in the area of software evolution. The proposed technique helps to reduce the occurrence of the fragile pointcut problem by eliminating the need for signature-based pointcuts for binding object references used in advice. The technique also reduces, although it does not entirely eliminate, the dependency between a role annotation and the method signature. Role annotations are written using expressions in the application's data model, rather than the AspectJ pointcut expression model. This may allow refactoring tools to more easily maintain the role annotations in the presence of application software evolution. Finally, the proposed technique eliminates some of the boilerplate code normally written by the programmer to implement the crosscutting data model of an AOP framework.

5. REFERENCES

- [1] J. Hannemann. *Role-Based Refactoring of Crosscutting Concerns*, Ph.D. Thesis, The University of British Columbia, 2005.
- [2] S. Hanenberg, M. Al-Mansari, R. Unland. Aspect-Specification Based on Structural Type Information, *Proceedings of the 2006 Symposium on Applied Computing*, France, 2006.
- [3] S. Hanenberg and A. Schmidmeier, Idioms for Building Software Frameworks in AspectJ. In: AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, 2003.
- [4] A. Kellens, K. Gybels, J. Brichau, K. Mens. A Model-driven pointcut language for more robust pointcuts. In *Software Engineering Properties of Languages and Aspect Technologies (SPLAT)*, 2006.
- [5] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *European Conference on Object-Oriented Programming, ECOOP 2005*, 2005.
- [6] C. Koppen and M. Stoerzer. Pcdiff: Attacking the fragile pointcut problem. In First European Interactive Workshop on Aspects in Software (EIWAS), 2004.
- [7] K. Lieberherr, D. Lorenz, J. Ovlinger. Aspectual Collaborations: Combining Modules and Aspects, *The Computer Journal*, 46(5) 2003.
- [8] D. Palmer, M. Kirschenbaum, L. Seiter, Emergence-Oriented Programming, *IEEE SMC 2005, Inter. Conf. on Systems, Man and Cybernetics*. October 2005.
- [9] L. Seiter, D. Palmer, M. Kirschenbaum, An Aspect-Oriented Approach for Modeling Self-Organizing Emergent Structures, In *International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS)*, 2006.
- [10] M. Stoerzer 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.
- [11] B. Vanhaute, B. De Win and B. De Decker. Building Frameworks in AspectJ, ECOOP 2001 Workshop on Advanced Separation of Concerns, Budapest, 2001.
- [12] <http://caesarj.org>
- [13] <http://www.objectteams.org>
- [14] <http://www.aspectj.org>
- [15] <http://java.sun.com/j2se/1.5.0/docs/guide/apt/index.html>
- [16] <http://jakarta.apache.org/commons/jexl/>