2009

Contributing Factors to Pointcut Fragility

Phil Greenwood
Relative Insight

Awais Rashid
University of Bristol

Raffi T. Khatchadourian
CUNY Hunter College

Contributing Factors to Pointcut Fragility

Phil Greenwood, and Awais Rashid
Lancaster University, UK
{greenwop, marash}@comp.lancs.ac.uk

Raffi T. Khatchadourian
The Ohio State University, USA
khatchad@cse.ohio-state.edu

Abstract

Pointcut fragility is a well-documented problem of Aspect-Oriented Programming with changes to the base-code causing join points to incorrectly fall in or out of scope. In order to combat this problem a tool was developed that provides mechanical assistance to pointcut maintenance. This tool relied on the deep structural commonalities between program elements to detect when pointcut fragility occurs. During the assessment of this tool a number of common practices were uncovered that were employed both in the aspect and base-code that contributed to or prevented pointcut fragility. This paper documents the practices uncovered and describes how they can affect pointcut fragility and design stability in general.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Metrics; D.1.0 [*Programming Techniques*]: General

General Terms Design, Experimentation, Measurement

Keywords adoption, assessment, metrics, modularity

1. Introduction

Aspect-Oriented Programming (AOP) [1] has emerged with the aim of reducing the scattering and tangling of crosscutting concerns (CCCs). This is achieved through the notions of advice being applied at join points identified by pointcuts expressions (PCEs). However, AOP is not without its problems or detractors.

One of the more well-documented problems is that of pointcut fragility. Consider an example PCE `execution(* m*(..))` which selects the execution of all methods whose name begins with *m*, taking any number and type of arguments, and returning any type of value. Suppose that in a particular version of the base-code, the above PCE selects the correct set of join points in which a CCC applies. As the software evolves, this set of join points may change as well. We say that a PCE is *robust* if it, in its unaltered form, is able to *continue* to capture the correct set of join points in future versions of the base-code. Thus, the PCE given above would be considered robust if the set of join points in which the CCC applies *always* corresponded to executions of methods whose name beings with *m*, taking any number and type of arguments, and so forth. However, with the requirements of typical software tending to change over time, the corresponding source code may undergo many alterations to accommodate such change, including the addition of *new* elements in which existing CCCs should also apply. Without *a priori* knowledge of future maintenance changes and additions, creating robust PCEs is a daunting task. As such, there may easily exist situations where the PCE itself must evolve *along* with the base-code; in these case we say that the PCE is *fragile*. Hence, the *fragile pointcut problem* [2] manifests itself in such circumstances where join points incorrectly fall in or out of the scope of PCEs.

In order to reduce the problems associated with pointcut fragility the authors of this paper developed a tool named "Rejuvenate Pointcut" [3]. The main premise behind this tool involves the creation of concern graphs [4] which represent the underlying program by identifying program entity types (i.e., classes, methods, fields, etc.) and the relationships between them (i.e., contains type, declares method, calls method, gets field, etc.). Concern graphs which pass through every join point shadow (to a pre-configured depth) are created and extracted from the base-code. These concern graphs can then be compared to extract common patterns between join points which are quantified by the same PCE. A confidence value can then be associated with each pattern which determines how representative the pattern is of the set of join points analysed. These patterns can then be applied to future versions of the base-code to identify potential join points where the same patterns occur. The suggested join points then also have a confidence value associated to them which allows a ranked list of suggested join points to be offered to the developer.

The evaluation of Rejuvenate Pointcut involved applying it to 23 publicly available AspectJ benchmarks, applications and libraries of varying size and domains. This evaluation was divided into two phases, the first phase involved assess-

OOPSLA'09, October 25–29, 2009, Orlando, Florida, USA.
ACM 978-1-60558-220-7/08/10.

ing the unique structural commonality typically portrayed by join point shadows selected by a single PCE by studying the attributes of patterns extracted from a single version of the AspectJ programs. The second phase involved assessing the accuracy of the suggestions offered by Rejuvenate Pointcut. This required comparing the list of suggested join points with those now included in the (human) revised version of the PC in a subsequent software version. This second phase involved analysing a sub-set (four) of the original 23 systems which contained multiple-versions that were created through applying a series of maintenance changes.

These results allow potential fragile pointcuts to be identified and point to practices employed in both the aspect and base-code that can contribute to pointcut fragility. The purpose of this paper is to document the findings of this analysis and detail how these discovered practices contribute to pointcut fragility and design stability in general.

The remainder of the paper is structured as follows. Section 2 outlines the practices identified during the evaluation of Rejuvenate Pointcut. The instances where pointcut fragility issues were caused by these practices are described in Section 3. Subsequently, Section 4 outlines the threats to validity of these observations. Finally, Section 5 offers some concluding remarks.

2. Employed Practices

The results presented in [3, 5] show the usefulness and accuracy of the Rejuvenate Pointcut approach. These results pointed to specific join points and pointcut expressions which were potentially affected by pointcut fragility issues. These points were identified through analysing Rejuvenate Pointcut results and determining the join points which incorrectly fell in or out of scope of a pointcut expression. The associated pointcut expression and the practices used around the advised join points are examined to determine their influence on pointcut fragility.

This section attempts to generalise the causes of pointcut fragility but also provide concrete examples of these causes occurring. The examples are extracted from just two of the systems analysed: HealthWatcher and MobilePhoto for space and consistency reasons but instances of these problems occurred in all systems. Furthermore, both of these systems have previously been used in other studies relating to design stability [6, 7] and so this previous analysis can provide insights for our study.

A number of practices have been identified in all analysed systems that both contribute towards reducing and increasing pointcut fragility. We will subsequently analyse these practices further to determine their direct affect on pointcut fragility.

2.1 Conflicting Effects of Marker Interfaces

A common practice in AO applications, and used extensively within HealthWatcher and MobilePhoto is declaring pointcuts in terms of an interface introduced within the aspect itself. This interface is generally used as a marker (i.e., declares no methods to implement) to identify classes where a CCC should be applied (see Figure 1 line 3). This interface is then composed via the AspectJ declare parents statement which enumerates the relevant classes (see Figure 1 line 5). The PCE can then be expressed in terms of the marker interface (see Figure 1 line 7), this allows generic PCEs to be defined that intercept constructor calls to any of the classes on which the interface has been composed or any methods within those classes that follow a certain naming convention. This can be seen as a common technique employed to reduce the number of changes necessary when an aspect driven change occurs, i.e., when the scope of CCC needs to be increased or reduced.

Although the same end effect can be achieved without using this marker-interface based approach, by naming each individual class within the PCE, marker interfaces do reduce the length of the PCE with the actual pointcut only specified in terms of one entity (i.e. the introduced interface). However, it can be argued that this approach does increase the complexity by introducing an extra composition statement, in that the declare parents statement is needed in addition to the pointcut statement which composes the crosscutting behaviour. This can be argued to increase the level of indirection which causes extra complexity when understanding the aspect behaviour. This is particularly problematic when the concern is spread over a number of aspects (i.e., an abstract aspect and a concrete aspect). One of the reasons for introducing an interface and specifying a PCE in this manner is to overcome differences between the underlying classes and allow the PCE to be defined using a common artefact. However, this practice causes a set of seemingly unrelated classes to be associated together which should be applied with caution. Due to the possibility of the classes being completely disassociated and originating from different sources (i.e., different concerns) they could be subject to different maintenance changes. These changes could then ripple through the system and cause pointcut fragility issues. Furthermore, this technique relies upon all advised classes being able to be treated the same (i.e., the same advice applied).

An example of this technique occurred within the analysed systems with the Synchronization and Persistence aspects (two typical CCCs) implemented using this practice. In both cases, the classes which these aspects apply to relate to different elements of functionality. This demonstrates the benefits of this approach, in that a concise PCE can be created despite the significant differences between the classes. However, it does cause issues as, in the case of the Persistence aspect, each class cannot be advised in the same way. Within the Persistence advice (see Figure 1 lines 7 to 15) each advised type needs to be determined at run-time via a series of if-statements to invoke the correct behaviour. This practice shifts the fragility issues from being contained

```
1  public aspect Persistence{
2
3    private interface SystemRecord {};
4
5    declare parents: Complaint || MedicalSpeciality ||
         HealthUnit implements SystemRecord;
6
7    Object around(): call(SystemRecord+.new(..)) ! within(
         Persistence+){
8      Class type= thisJoinPoint.getSignature().
           getDeclaringType());
9      if (type.equals(Complaint.class)) {
10       return new ComplaintRecord(getComplaintRepository()
             );
11     } else  if (type.equals(HealthUnit.class)) {
12       return new HealthUnitRecord(getHealthUnitRepository
             ());
13     } else  if (type.equals(MedicalSpeciality.class)) {
14       return new MedicalSpecialityRecord(
             getSpecialityRepository());
15   }
```

Figure 1. Persistence aspect using a marker interface.

within the PCEs to within the body of the advice which is potentially more problematic. In the case of the Synchronization aspect, the same advice can be be applied to all advised join points and so does not suffer from the same fragility issues.

Although there are cases whereby other concerns are composed via interfaces, these are two cases where a marker interface is introduced. The other cases utilise PCEs that are specified in terms of interfaces that are already a part of the base-code and so the interface do not act as markers to compose the aspect. Also in these cases, the classes implementing the interface have a genuine relationship with each other beyond requiring some CCC (i.e., they implement some closely-knit functionality). Furthermore, these interfaces tend not to be "empty", in that they define some methods which the classes implementing the interface must define. This creates an additional relationship between the classes which can be utilised within the PCE and so potentially reduces pointcut fragility.

2.2 Pointcuts with Except Clauses

In order to overcome pointcut fragility and ensure PCEs are widely applicable and generalised enough to cope with basic maintenance changes, wild-cards are often utilised to take advantage of naming conventions used within the base-code. However, a problem with utilising wild-cards in this manner is that they can often be too widely scoped and advise unintended join points. To prevent these problems, *except* clauses are often added to PCEs to exclude the unnecessary join points. This causes the PCE to contain multiple primitive pointcut designators that all need to be evaluated to correctly compose the aspect.

In many ways PCEs which contain an except clause are similar to those which use marker interfaces (Section 2.1) in that the commonalities which are being exploited are not universally prevalent. In the case of the marker interfaces

```
1  pointcut clientDistribution(): call(* IFacade+.*(..))
       && ! call(static * *.*(..));
```

Figure 2. Example of a homogeneous crosscutting concern PCE with an except clause.

```
1  pointcut initMenu(MediaListScreen screen):
2    execution(public void MediaListScreen.initMenu()) &&
       this(screen);
```

Figure 3. PCE relating to a heterogeneous concern.

the commonalities are entirely fabricated, however, the commonalities exploited in PCEs which contain except clauses are only partially fabricated. In these PCEs a pattern has been found that fits most, but not all, cases (see Figure 2). By including except clauses, this pattern is being manipulated and artificially fabricated to enable the CCC to "fit" the base-code.

Although in a single version system this approach will not cause any problems, apart from being an unwieldy PCE, when the base-code is evolved PCEs which employ this practice can be susceptible to fragility problems. This is due to the PCE being dependent on multiple parts of the base-code to remain consistent and applicable in terms of the advised join points due to it containing multiple primitive pointcut designators. If any of the advised shadows are altered to fall out of scope of the PCE, then the PCE needs to be updated. However, there is now a possibility of non-advised shadows falling out of scope of the except clause causing it to be incorrectly advised. Similarly, a newly added segment of code has to satisfy both parts of PCE for it be correctly advised. Each extra primitive designator of the PCE that has to be satisfied for a join point shadow to be correctly advised or not advised increases the potential for fragility.

2.3 Heterogeneous vs. Homogeneous CCCs

In Aspect Oriented Software Development (AOSD) there is often a distinction made between heterogeneous and homogeneous CCCs [8]. A heterogeneous CCC advises distinct join points with unique advice (see Figure 3), this requires PCEs to precisely quantify the join point they advise. Whereas a homogeneous CCC advises multiple join points with the same advice (see Figure 2), this requires generalised PCEs to quantify all join points to advise. Both types of CCC occur in HealthWatcher and MobilePhoto. However, it is not clear which type of CCC is more susceptible to pointcut fragility.

From one perspective heterogeneous CCCs could be viewed as being more susceptible to pointcut fragility due to the tight-coupling between the PCE and the base-code. Any changes made to the base-code is likely to propagate to the aspect making it likely that the PCE will need to be

updated. This is because the PCE is unlikely to utilise wild-cards, making the PCE very rigid to ensure that the concern is applied to the desired narrow scope. However, due to the narrow scope of where the advice is applied, there are fewer reasons for change, in that any change would have to target the specifically advised join point.

Alternatively, a homogeneous CCC is more widely scoped. This increases the potential for changes to affect an advised join point which may propagate to the aspect and may expose any pointcut fragility. However, as the concern is more widely scoped it is likely that the PCE is more generalised causing it to be more flexible, allowing it to absorb any minor changes which would otherwise cause pointcut fragility.

One reason for making a change to a PCE is to alter the composition scope of the aspect (i.e., the scope needs to be narrowed or widened). The applicability of this type of change to heterogeneous CCCs is questionable, as the concern is often of a very specific nature, so it is unlikely that it will be applicable to other join points in the future. This makes it much more likely that the reason for changing a PCE related to a heterogeneous CCC is pointcut fragility. However, this theory will have to be verified when observing the actual pointcut fragility which occurred within HealthWatcher and MobilePhoto.

3. Observed Pointcut Fragility

The aim of the previous section was to extract and comment on some of the uses of AO techniques that may contribute to or reduce pointcut fragility the purpose of this section is to observe how pointcut fragility manifests itself and how these techniques actually contribute to the fragility.

3.1 Refactoring and Pointcut Fragility

One of the most significant causes of pointcut fragility, which was observed within both HealthWatcher and MobilePhoto, is refactoring changes which are applied to the base-code. This was most evident in HealthWatcher where the majority of the maintenance changes involved performing refactoring changes to improve the overall design of the system. However, there was also evidence of refactoring changes causing pointcut fragility in MobilePhoto.

One instance where the effect of this type of change is evident involved introducing the Command pattern [9] to HealthWatcher. The introduction of the Command pattern involved relocating the implementation of a set Servlets' implementation to a series Command pattern classes. A number of PCEs defined within HealthWatcher were dependent on the signature of the Servlets to apply a number of CCCs, including Distribution and Exception Handling. These refactoring changes caused these PCEs to be invalidated. However, due to the nature of the refactoring changes the scale of the fragility problems were quite minor. Only a minor correction to the PCEs was needed which involved altering the pattern to quantify over Command classes rather than Servlet classes. As the core behaviour of these classes remained the same no further correction was necessary.

Although the aim of the maintenance changes were slightly different in MobilePhoto, with their aim being to introduce a series of optional and mandatory features, a number of refactoring changes also had to be applied. For example, when adding support for a new media type, the existing data types had to be generalised to support both the existing and new data types. This involved creating a new abstract type which the existing and new data types could inherit from. However, a number of PCEs were specified only in terms of the existing types and so the new types would still incorrectly fall-out of scope despite having a common inheritance structure with the existing types. Instead the PCE had to be modified to now quantify over the abstract type and so allow both the new and existing data types to be advised. As with the HealthWatcher example this involved minor, but wide-spread, change to the representation of the PCEs.

Interestingly, almost none of the PCEs specified within MobilePhoto contained wild-card tokens. Instead each of the PCEs were precisely specified to ensure the desired join points were advised (see Figure 3). A consequence of this is that none of the PCEs need an except clause as the precision of the PCE does not cause any unintended join points to be advised. In comparison, HealthWatcher has a much larger number of PCEs that contain wild-card tokens (see Figure 2). However, both approaches are equally susceptible to pointcut fragility and so the generalised vs. precise nature of PCEs is not a significant factor in terms of the fragility of the PCEs when performing refactoring changes. This is due to PCEs which contain wild-card tokens typically generalise either method names or parameter types and not the class names or packages. In our observations, it is normally *all* of these elements which are affected by refactoring and so the generalisation applied is unable to absorb such changes.

From these findings we can can conclude that although refactoring changes to the base-code are a significant cause of pointcut fragility, the effect they have in terms of the necessary corrections are relatively minor and are normally easily fixed (i.e. a simple renaming).

3.2 Extensions and Pointcut Fragility

The primary purpose of the maintenance changes applied to MobilePhoto are to add new features. This would normally be expected to expose pointcut fragility problems as CCCs need to be applied to the newly introduced features and so PCEs need to be updated to the compose the concerns accordingly. However, beyond the refactoring cases mentioned in Section 3.1 this did not occur in MobilePhoto. One of the major contributing factors that prevented this is the software product line nature of MobilePhoto. A lot of the features introduced during the maintenance changes were either optional or alternative features meaning that they had to be added as unobtrusively as possible to allow them to be easily omitted or included as the customer specification dic-

tates. Therefore, this strategy reduced the need for changes to made to PCEs.

In contrast, when extensions were applied to HealthWatcher to introduce new features, pointcut fragility did become an issue. As expected the use of marker interfaces (see Section 2.1) did allow some pointcut fragility issues to be absorbed. Rather than having to quantify over all the newly introduced join point shadows, only the declare parents statement had to be updated to enumerate the new classes and the existing PCE will then correctly identify the new join point shadows. Although this is still classed as pointcut fragility, the use of marker interfaces minimises the effect of this fragility. Instead of a major change to the PCE having to be made, a relatively minor change of simply adding the new class which should be advised to the declare parents statement. Although the correction to the PCE is minor, more extensive changes had to be made to the advice body compared to if the marker interface was removed and the advised classes quantified individually. The marker interface PCE causes the same advice to be applied to multiple types, this is fine when the advised type is irrelevant and the same behaviour can be applied to all types. However, when different behaviour has to be applied to different types this causes problems as occurred with the persistence aspect in HealthWatcher (see Figure 1). Within the advice body a series of if-blocks are used to determine the currently advised type and invoke the correct behaviour. When new types need to be advised, as occurred when extensions were added during the maintenance changes applied, this advice body needs to be updated to include the relevant if-blocks. Although this is not technically a case of pointcut fragility, it is a consequence of the PCE specified and can be attributed to pointcut fragility issues propagating through the aspect.

In the previous sub-section we mentioned that in terms of refactoring changes the fragility of the pointcut is not affected by the use of wild-card tokens. However, the same cannot be said for changes that involve extending existing functionality. The use of wild-card tokens to create generalised PCEs appear key to prevent pointcut fragility. As part of the maintenance changes applied to HealthWatcher, a number of new operations had to be added that were similar to existing operations, which have advice composed to them, but operated on different types. In these cases it was necessary to apply the same advice to the new operations. The generic nature of the PCEs enabled these new operations to be advised without modification. In contrast, the precise PCEs specified in MobilePhoto were subject to fragility issues caused by the refactoring activities that were necessary to introduce the new features.

3.3 Heterogeneous to Homogeneous Concerns

As stated earlier in Section 2.3 there is a significant difference between heterogeneous and homogeneous CCCs in terms of the scope of the join point shadows which they advise. These differences were made apparent when the maintenance changes were applied due to the different observable effects on each type.

Firstly, homogeneous CCCs are more susceptible to pointcut fragility issues. All the concerns discussed so far have all centred on PCEs that relate to homogeneous concerns and none relate to heterogeneous concerns. This can be attributed to the fact that homogeneous concerns are more prevalent throughout the base-code examined and so the chances of a maintenance change affecting a shadow advised by a homogeneous concern is much greater.

Secondly, although the frequency of pointcut fragility occurring within heterogeneous advice is much less, the effects on them are much more pronounced particularly when the heterogeneous CCC is not a true heterogeneous concern. This situation occurred within HealthWatcher whereby a Synchronization aspect which utilised a concurrency manager advised a single type and so contained PCEs that quantified over a series of independent shadows causing it to be classified as a heterogeneous concern. However, during the course of the study it was discovered that this concern was also applicable to other types which were introduced as a result of the maintenance changes. This required converting the aspect from being a heterogeneous concern to be a homogeneous concern. However, the initial state of the aspect meant that it was highly coupled to the original type which it was advising meaning that extensive changes had to be made to allow multiple types and shadows to be advised. This involved not only making significant changes to the PCEs, but also other changes regarding the whole structure of the aspect to support multiple types.

The product line nature of MobilePhoto caused it to have several heterogeneous CCCs. Implementing optional and alternative features in this way allows them to be easily omitted or included in a particular configuration, however, these features are normally very specific to a particular part of the base-code causing them to be heterogeneous concerns. There are only a small number of homogeneous concerns present in MobilePhoto the most significant being Exception Handling which was frequently updated throughout the maintenance changes. These observations confirm the findings from HealthWatcher that it is the homogeneous concerns that are most susceptible to pointcut fragility even though heterogeneous concerns are more prevalent in MobilePhoto. Again, the findings from MobilePhoto confirm that heterogeneous concerns are likely to require more extensive changes to correct pointcut fragility issues. A concern was also observed in MobilePhoto which was thought to be heterogeneous and to be extensively modified to advise additional types due to the maintenance changes applied.

4. Threats to Validity

The causes and techniques to reduce pointcut fragility identified above are not intended to be a definitive lists and other phenomenon may be identified in other systems or applied

AO approaches. A number of factors associated with this study have been identified which could limit the scope of fragility issues uncovered.

Firstly, the number of available multi-version AO systems with a range of non-trivial CCCs limits the scope of the study. However, the systems analysed do contain a broad-range of non-trivial CCCs and so are representative of AO software. In the future we hope to obtain a larger range of applications to enable a broader study to be performed to establish whether the practices employed and causes of pointcut fragility are relevant to a number of domains.

A subsequent factor of this, in terms of the available systems, is that it reduces the scope of potential changes in that they may not test all PCEs within each system. However, the changes analysed were applied prior to this study being conceived and so do not have any inherent bias towards exposing or hiding pointcut fragility. Furthermore, the maintenance changes applied cover a range activities which are realistic in terms of maintenance tasks typically performed in real-world software development environments.

Finally, it can be argued that some of the cases where we have observed pointcut fragility could have been avoided by implementing the base-code, aspect-code or the maintenance changes in an alternative fashion. However, we argue that it is not our place to comment on the quality or design decisions made to implement the various artefacts we have studied. Our aim was to observe the strategies employed and comment on whether these choices increased or reduced the pointcut fragility. By documenting these choices and strategies employed we hope to allow other developers to avoid similar pitfalls.

5. Summary

This paper has analysed the phenomenon of pointcut fragility observed within a number of AspectJ systems. The evaluation of pointcut maintenance tool required the deep analysis of these systems which uncovered a series of commonly employed practices which both contribute and prevent pointcut fragility. Two systems were selected for more rigorous analysis: HealthWatcher and MobilePhoto. The practices identified that affect pointcut stability include:

- The use of marker interfaces.
- Pointcuts with multiple primitive pointcut designators.
- Heterogeneous and homogeneous concern differences.

These techniques were then analysed to determine their true effect on pointcut fragility within both HealthWatcher and MobilePhoto. From this a number of common causes of pointcut fragility that go beyond blaming pointcut fragility on the naming conventions used in PCEs were uncovered:

- Introducing marker interfaces to identify classes to advise reduces the extent of changes to PCEs but causes ripple-effects through the associated aspect.
- Changes which affect heterogeneous CCCs cause significant changes to both the PCE and aspect implementation.
- Refactoring maintenance changes frequently cause pointcut fragility issues.
- PCEs containing multiple pointcut primitive designators do not seemingly affect pointcut fragility.
- Applying extensions to existing behaviour relies on respecting established practices to reduce pointcut fragility.

Although these practices and their effects are not a definitive list and others are highly likely to exist, they do provide some initial insights to both AO system developers and researchers developing new aspect-oriented techniques. These practices need to be more formally analysed and their effect on pointcut stability quantitively measured. However, they highlight potential problems to AO system developers and allow them to take action to prevent the pointcut fragility and stability issues from occurring and so improve the quality of their code. They also provide a road-map to researchers and allow them to develop techniques to minimise the effects these practices have.

References

[1] Gregor Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect oriented programming. In *ECOOP*, 1997.

[2] C. Koppen and M. Stoerzer. PCDiff: Attacking the fragile pointcut problem. In *Eur. Int. Workshop on Aspects in Software*, 2004.

[3] Raffi Khatchadourian, Phil Greenwood, Awais Rashid, and Guoqing Xu. Pointcut rejuvenation: Recovering pointcut expressions in evolving aspect-oriented software. In *to appear in IEEE/ACM International Conference on Automated Software Engineering (ASE 09)*, 2009.

[4] Martin P. Robillard and Gail C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *ICSE*, 2002.

[5] Raffi Khatchadourian, Johan Dovland, and Neelam Soundarajan. Enforcing behavioral constraints in evolving AO programs. In *FOAL*, 2008.

[6] Phil Greenwood and et al. On the impact of aspectual decompositions on design stability: An empirical study. In *ECOOP*, 2007.

[7] Eduardo Figueiredo and et al. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE*, 2008.

[8] Charles Zhang and Hans-Arno Jacobsen. Efficiently mining crosscutting concerns through random walks. In *AOSD*, 2007.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.