Detecting Broken Pointcuts Using Structural Commonality and Degree of Interest

Raffi Khatchadourian[a,1,*], Awais Rashid[b], Hidehiko Masuhara[c,2], Takuya Watanabe[d,3]

[a]*Department of Computer Science, Hunter College, City University of New York, 695 Park Avenue, Room 1008, Hunter North Building, New York, NY 10065 USA*
[b]*School of Computing and Communications, Lancaster University LA1 4WA, UK*
[c]*Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, Tokyo 152-8552, Japan*
[d]*Edirium K.K., 1-8-21 Ginza, Chuo, Tokyo 104-0061, Japan*

Abstract

Pointcut fragility is a well-documented problem in Aspect-Oriented Programming; changes to the base-code can lead to join points incorrectly falling in or out of the scope of pointcuts. Deciding which pointcuts have broken due to base-code changes is a daunting venture, especially in large and complex systems. We present an automated approach that recommends pointcuts that are likely to require modification due to a particular base-code change, as well as ones that do not. Our hypothesis is that join points selected by a pointcut exhibit common structural characteristics. Patterns describing such commonality are used to recommend pointcuts that have potentially broken with a degree of confidence as the developer is typing. The approach is implemented as an extension to the popular Mylyn Eclipse IDE plug-in, which maintains focused contexts of entities relevant to the task at hand using a Degree of Interest (DOI) model. We show that it is accurate in revealing

*Corresponding author.
Email addresses: raffi.khatchadourian@hunter.cuny.edu (Raffi Khatchadourian), awais@comp.lancs.ac.uk (Awais Rashid), masuhara@acm.org (Hidehiko Masuhara), sodium@edirium.co.jp (Takuya Watanabe)

[1]A portion of this work was administered during his visit to the Department of Graphics and Computer Science at the University of Tokyo.

[2]A portion of this work was administered while he was with the Department of Graphics and Computer Science, Graduate School of Arts and Sciences, University of Tokyo, Tokyo, Japan.

[3]A portion of this work was administered while he was with the Department of Graphics and Computer Science, Graduate School of Arts and Sciences, University of Tokyo, Tokyo, Japan.

broken pointcuts by applying it to multiple versions of several open source projects and evaluating the quality of the recommendations produced against actual modifications. We found that our tool made broken pointcuts 2.14 times more interesting in the DOI model than unbroken ones, with a p-value under 0.1, indicating a significant difference in final DOI value between the two kinds of pointcuts (i.e., broken and unbroken).

Keywords: Software development environments, Software maintenance, Software tools

Highlights

- An automated approach that recommends pointcuts that likely have or have not broken due to a particular base-code change is proposed.
- The approach relies on join points selected by a pointcut exhibiting common structural characteristics.
- Commonality patterns are used to recommend possibly broken pointcuts with a degree of confidence as the developer is typing.
- The approach was implemented as an extension to the Mylyn Eclipse IDE plug-in, dynamically manipulating its Degree of Interest (DOI) model.
- Broken pointcuts were 2.14 times more "interesting" than unbroken ones, with a p-value under 0.1.

---

1. Introduction

Although using Aspect-Oriented Programming (AOP) [1] can be beneficial to developers in many ways [2, 3, 4, 5], such systems have the potential for new problems unique to the paradigm. A key construct that allows code to be situated in a single location but affect many system modules is a query-like mechanism called a pointcut expression (PCE). PCEs specify well-defined locations (join points) in the execution of the program (base-code) where code (advice) is to be executed. In AspectJ [6], an AOP extension of Java, join points may include calls to certain methods, accesses to particular fields, and modifications to the run time stack. In this way, AOP allows for localized implementations of so-called crosscutting concerns (or aspects), e.g.,

logging, persistence, security. Without AOP, aspect code would be scattered and tangled with other code implementing the core functionality of the modules.

As the base-code changes with possibly new functionality being added, PCEs may become invalidated. That is, they may fail to select or inadvertently select new places in the program's execution, a problem known as pointcut fragility [7]. As an example, consider the PCE **execution**(∗ send∗(String)) for a security aspect that selects the execution of all methods whose name begins with *send*, taking a single String parameter, and returning any type of value, with the intent of encrypting outbound messages. Suppose that in a particular version of the base-code all methods that send messages have names that match this pattern. In other words, in this version, this PCE selects and only selects the correct set of join points to which this aspect applies. Now suppose that in a subsequent version a new method is introduced that also sends messages but whose name begins with *transmit*. In this case, the PCE is fragile as it fails to select the execution of this new method, which also requires encryption.

Deciding which PCEs have broken is a daunting venture, especially in large and complex systems. In software with many PCEs, seemingly innocuous base-code changes can have wide effects. To catch these errors early, developers must manually check all PCEs upon base-code changes, which is tedious (potentially distracting developers), time-consuming (there can be many PCEs), error-prone (broken PCEs may not be fixed properly), and omission-prone (PCEs may be missed).

1.1. Languages and Other Mechanisms for Coping with Pointcut Fragility

Several approaches combat this problem by proposing new PCE languages with more expressiveness [8, 9, 10, 11, 12, 13, 14], limiting where advice may apply [15, 16], or enforcing constraints on advice application [17, 18, 19, 20]. Others make advice applicability more explicit [21] or do not use PCEs [22, 23, 24]. However, each of these tends to require some level of anticipation and, consequently, when using PCEs, there may nevertheless exist situations where PCEs must be manually updated. Furthermore, when using more expressive PCE languages, the rules that the base-code must respect may be complex. Hence, although these languages may reduce fragility, they may render *detection* of broken PCEs more difficult [25].

Programmer-defined source code annotations [26] can also be used to "mark" relevant locations where a crosscutting concern (CCC) applies. PCEs then use these annotations to accurately select the appropriate join points. If used properly, i.e., if all locations where the CCC applies are correctly annotated and if the corresponding PCE correctly selects these elements, this scheme can produce PCEs that are robust to changes such as refactorings since names and organization of program elements

may change but the associated annotations remain intact. However, refactoring is not the only reason a PCE breaks. For example, adding a new element but neglecting to annotate it properly with *all* CCCs that apply to it will break an annotation-based pointcut.

1.2. Tool-support for Detecting Broken Pointcuts

Other approaches offer tool-support for detecting broken PCEs. The AspectJ Development Tools (AJDT) [27], which displays current join point and PCE matching information, does not indicate which PCEs do *not* select a given join point nor which are likely broken due to a new join point. [7] discovers PCEs that exhibit differences in advice application, however, PCEs that contain no join point changes between base-code versions, e.g., when new system functionality is added, may also be broken. [28] augments the AJDT with *almost matching* join point information by relaxing PCEs using developer-minded heuristics but do not detect situations where join points are unintentionally selected by PCEs. [29] automatically fixes PCEs broken by refactorings, however, manual base-code edits may also break PCEs. [30] determines which PCEs are *syntactically* affected by new join points but does not necessarily identify *semantic* breakages. [31] suggests join points that may require inclusion by a revised version of a PCE. Yet, developers must *manually* detect broken PCEs, as well as determine how frequently to check.

1.3. Broken Pointcut Detection Approach

In this article, we present an automated approach that recommends PCEs that are likely to require modification due to a particular base-code change. Our approach has been implemented as an AspectJ source-level inferencing tool called FRAGLIGHT, which is a plug-in for the popular Eclipse (`http://eclipse.org`) IDE. FRAGLIGHT identifies, as the developer is making changes to the base-code, PCEs that have likely broken within a degree of *change confidence*. Based on how "confident" we are in the PCE being broken, FRAGLIGHT presents the results to the developer by manipulating the Degree of Interest (DOI) model in an existing tool, i.e., Mylyn [32].

To the best of our knowledge, our approach is the first of its kind to integrate with Mylyn and manipulate its DOI model based on change prediction/impact analysis. Mylyn [33] is a standard Eclipse plug-in that facilitates software evolution by focusing graphical components of the IDE in order that only ("interesting") artifacts related to the currently active task are revealed to the developer [34]. Mylyn works by maintaining and manipulating a DOI model as the developer works on the project. The more interaction a developer has with a particular artifact (e.g., a file), the more prominent it appears in the IDE, and the less prominent other less recently used artifacts appear.

Our approach enables developers to discover problematic PCEs early in development so that they may be fixed before causing bugs that may compound over time. FRAGLIGHT alleviates much of the burden associated with identifying broken PCEs, making these systems easier to maintain. We also show how a recommendation system can be seamlessly integrated into Mylyn, paving a way for future researchers to consider such integration. The potential of this approach, independent of AOP, is that the results of recommendation systems for software engineering can be brought to the forefront of developers in order that they can make practical use of them as soon as they write the code. This can be particularly useful for other change impact approaches.

Our key contributions can be summarized as follows:

Algorithm design. We present an automated approach that programmatically manipulates the Mylyn DOI model to bring broken PCEs to the base-code developer's attention early. The developer is informed, with a subtlety that varies on likelihood, when their code is likely to break PCEs as it is being written.

Mylyn integration. FRAGLIGHT implicitly makes PCEs that are *more* likely broken *more interesting*, i.e., by *increasing* its DOI value, while implicitly making PCEs that are *less* likely broken *less interesting*, i.e., by *decreasing* its DOI value. In this way, possibly broken PCEs are presented to the developer in a variably invasive way. In other words, PCEs that likely need developer attention are presented more prominently in the IDE than ones that are less likely. Developers can then make alteration decisions based on FRAGLIGHT's recommendations, possibly adjusting the PCE or the base-code to rectify the problem. Moreover, we pave an avenue for future research to also utilize Mylyn integration.

Implementation and experimental evaluation. To ensure real-world applicability, we implemented our approach as a seamless, publicly available extension to Mylyn. A study on 14 version changes consisting of 5,711 base-code edits of AspectJ programs indicates that the technique is effective and practical in detecting broken PCEs as the base-code developer types. Upon completion of the experiments, the average DOI value of PCEs that *actually* broke were, on average, 2.14 times greater than that of PCEs that *did not* break throughout versions, with a p-value under 0.1 indicating a significant difference in final DOI value between the two kinds of pointcuts (i.e., broken and unbroken). This demonstrates that using our approach results in broken PCEs being 2.14

times more prominently displayed in the IDE than unbroken PCEs,[4] bringing broken PCEs to the forefront while keeping unbroken PCEs in the background. These results advance the state of the art in automated tool-support for AOP evolution.

FRAGLIGHT's recommendations are based on harnessing unique and arbitrarily deep structural commonality between program elements corresponding to join points selected by a PCE in a particular software version. To illustrate, again consider the example PCE given earlier and suppose that, in a certain base-code version, the PCE selects the execution of three different message transmitting methods. Further, suppose that facets pertaining to these methods exhibit structural commonality. For instance, each of the methods' bodies may (textually) include a call to a common method `connect()` or an assignment to a common field `queue`. Likewise, each method may be declared in class `Message`. The majority of program elements corresponding to join points selected by a PCE in one base-code version share such characteristics between them, and these relationships persist in subsequent versions [31]. In this article, we use this premise to detect broken PCEs on-the-fly. Note that this structure-based approach is relevant to even annotation-based PCEs as new join points may need to be annotated.

Though we have tackled some issues of pointcut fragility in our previous work [31], this work tackles a different problem (i.e., broken pointcut *detection*) and differs in the following ways:

Solves a different problem. Our previous approach, geared towards *aspect* developers,[5] periodically suggests join points that may require inclusion into a revised version of a PCE. Aspect developers may revise *PCEs*, possibly after *coarse-grained* base-code changes, depending on the provided *join point* suggestions. Our new approach, geared towards *base-code* developers, however, suggests *PCEs* that may have broken due to a single revision to the base-code. Here, we provide base-code developers with feedback following a series of related, *fine-grained* base-code changes that may have broke PCEs using a new, incremental algorithm. Base-code developers may then revise the *base-code* depending on the suggestion provided by this new approach or inform the aspect developer of the problem.

[4]Elements with a significantly low DOI value may be hidden from view in the IDE. The DOI value of 0 is used in the comparison of such elements.

[5]The distinction between aspect and base-code developers has been well-documented. This is particularly relevant in regards to reusable aspects [35].

Listing 1: A point on a Cartesian plane.

```
1 public class Point implements Figure {
2     private double x;private double y;
3     public void setX(double x) {this.x=x;}
4     public void setTwiceX(double x)
5         {this.x=2*x;}
6     public double getY() {return y;}}
```

Presents a new, incremental algorithm. While our previous approach works with only a single PCE at a time, in this article, our incremental approach avoids rebuilding and analyzing the base-code for each PCE. A new, incremental algorithm is developed to enable on-the-fly, broken PCE detection. A new *confidence* equation for *PCEs* is presented that corresponds to the probability that the PCE has broken due to a base-code change.

Integrates with Mylyn. Our new approach is integrally tied to the Mylyn DOI, a proven, successful, and familiar model.

A brief introduction of this work originally appeared in [36], and a demonstration of our preliminary tool, along with details of the implementation, appeared in [37]. In this article, we fully describe our approach and add an evaluation. The evaluation presents our experimental results to comprehensively and thoroughly assess the overall quality of the recommendations produced by our approach against actual modifications.

Organization

This article is organized as follows. In Section 2, we present a simple motivating example and discuss the associated challenges along with the limitations of related approaches. In Section 3, we discuss our approach to the problem and how it is integrated into Mylyn. We evaluate our approach on several real-world AspectJ projects and detail the results in Section 4. Section 5 contains more detailed information on related work. We conclude in Section 6, as well as set forth future work.

2. Motivating Example

We motivate our approach using a simple yet classic graphics application inspired by [6]. Though the example is small, we use its simplicity to detail our approach.

Listing 2: An aspect managing how Figures are displayed.

```
1 public aspect DisplayManipulation {
2   after():
3     execution(* Figure+.set*(..))
4       {Display.update();}
5   double around():
6     execution(double Figure+.get*(..))
7       {return proceed()*0.5;}}
```

Listing 1 portrays a code snippet of a simple Point class (line 1) that implements a Figure (interface not shown) on a Cartesian plane. Two instance fields, x and y, are declared on line 2. There are two mutator instance methods for field x (mutators for y omitted for presentation), namely, setX(**double**), declared on line 3, which assigns field x to be the argument, and setTwiceX(**double**), declared on line 4, which assigns field x to be double the argument. Furthermore, there is an accessor instance method for field y (accessor for x omitted for presentation), declared on line 6, that returns the field value.

As Figures may be maneuvered in many different editor modules, the DisplayManipulation aspect snippet (Listing 2) localizes the code for manipulating how Figures are displayed. The **after** advice (line 2) refreshes the Display (line 4, code not shown) whenever the state of a Figure is altered. This advice is implicitly executed **after** control leaves any join point selected by its bound PCE (line 3). These join points correspond to the **execution** of any method implementing a method of the Figure interface (Figure+) whose name begins with **set**, takes any number and type of parameters, and returns any type of value. In Listing 1, this corresponds to the execution of the setX(**double**) and setTwiceX(**double**).

Likewise, the **around** advice (line 5) scales Figures by 50%. The advice body (line 7) is implicitly executed *around* join points matching its bound PCE (line 6). Such join points correspond to the execution of methods implementing a method in the Figure interface whose name begins with **get**, taking any number and types of parameters, and returning any value. In Listing 1, this corresponds to the execution of the getY() method. When executed, the advice body first **proceed**s to execute the selected join point, multiplies the return value by the scaling factor, and returns the resulting value in its place.

Suppose that in this version, both PCEs are correct, i.e., they select all and only the intended join points. Now suppose that in a subsequent version, a new method move(**double**,**double**), which moves figures according to the specified coordinates, is

added to the Figure interface. A corresponding implementation is then added to the Point class:

Listing 3: A new method is added to move Figures using coordinates.

```
1 public void move(double x, double y)
2   {this.x=x; this.y=y;}
```

Clearly, this new method alters the state of Figures, however, the PCE bound to the **after** advice, which refreshes the Display following state changes to Figures, on line 3 of Listing 2 fails to select this new join point. As a result, this PCE breaks.[6] Notice, however, that the PCE bound to the **around** advice, which scales figures, does not break and thus continues to select all and only the desired join points.

In general, each incremental change to the base-code can potentially break PCEs and thus cause bugs. If developers wait until many such changes occur, problems may be compounded and more difficult to find. To alleviate this, developers could perform a global analysis of all aspects and verify that each PCE is correct after every incremental change. However, not only would such an activity be distracting to base-code developers, it could also be non-trivial. Although this simple example contains only two PCEs, larger, more realistic systems may contain much more PCEs whose correctness would need to be verified. It would thus be helpful for developers if broken PCEs could be brought to their attention early, even *before* continuous regression testing may catch them.[7] It would also be helpful if *un*broken PCEs were kept in the "background" as no action would be required. That way, the base-code developers may continue coding when an error is less likely and pause work otherwise. Rectifying such a problem would involve either changing the base-code[8] so that it is correctly selected (or not selected) by the problematic PCE, or by altering the PCE itself in the case that its source is available.

The problem described here is not easily addressed by existing techniques mentioned in Section 1. For example, the language support techniques discussed in Sec-

[6]This PCE could have instead selected field **set** join points, e.g., **set**(* Figure+.*), which would have seemingly solved the problem. However, interfaces (like Figure) do *not* contain variable instance fields, thus, such a PCE would not select the intended join points. Moreover, in the case of the Point class, the Display would have been refreshed twice, which could be inefficient. This would also be the case had the move() method implementation indirectly changed the coordinates via the setX() and setY() methods.

[7]Writing system tests that test the effects of advice is not always easy given that aspects normally affect large portions of the system that tend to crosscut module boundaries and represent non-functional concerns. Moreover, the affects of advice may be under specified [38].

[8]It may not always be possible to fix the problem using a base-code change as doing so may break other PCEs.

tion 1.1 may help prevent the problem from occurring by allowing for more expressive PCEs. If developers can correctly capture situations of where the crosscutting concern applies using more expressive PCEs, the problem may be circumvented. But, more expressive PCE languages do not offer assistance when the problem *does* occur. Moreover, developers may not be able to utilize a diverse set of different languages due to project constraints. The tool support techniques presented in Section 1.2 would not detect that the **after** advice *has* broken and that the **around** advice has *not* broken because the *new* move method does not match (or nearly match in terms of [28]) any existing PCEs, cause any differences in advice application, or result from a refactoring. In the following sections, we will demonstrate how FRAGLIGHT can automatically alleviate such problems.

3. Approach

3.1. Overview

A join point *shadow* (JPS) is the static counterparts of a join point, i.e., a point in the program text where the compiler may insert advice code [39]. FRAGLIGHT predicts how likely each PCE is to change given a change in the base-code. We model base-code changes as a series of JPS additions and removals, with each added JPS in the series being used as input.[9] Changing a JPS, e.g., renaming a method, is modeled as the addition of a new JPS, e.g., the new method's **execution**.

Example 1. Adding the move() method in Listing 3 would result in three new JPSs, namely, **execution**(**void** Point.move(**double**,**double**)), **set**(Point.x), and **set**(Point.y), with the latter two being on line 2 in Listing 3.

Figure 1 depicts Venn diagrams of four canonical situations and how FRAGLIGHT makes predictions in each. The numbers at the bottom right-hand corner of the diagrams correspond to the situation numbers below. We consider a single PCE, structural pattern, and input JPS to simplify the presentation. The universe is all of the program's JPSs. The region labeled *PCE* represents the set of all JPSs selected by the PCE. The region labeled $\hat{\pi}$ represents all JPSs corresponding to program elements matched by a particular *structural pattern*. Structural patterns depict organizational relationships between program elements, e.g., all methods declared by a class (a single depth pattern), all methods whose bodies textually contain a call to methods whose bodies include a statement that writes to a particular field (a multi-depth pattern). Again for simplification, Figure 1 shows a single structural pattern, whereas many structural patterns may exist in a given program.

[9] Note that removing a JPS can never break a PCE.

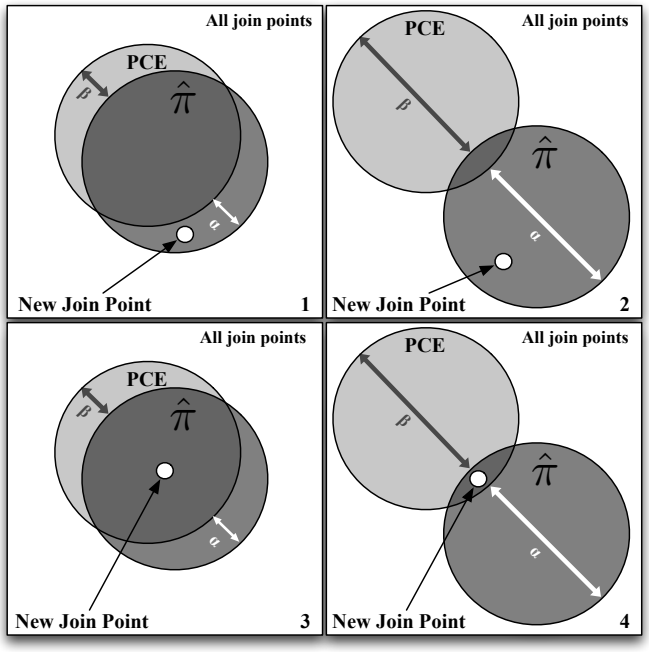


Figure 1: Predicting whether a PCE breaks due to a join point being added to the base-code. *PCE* represents all join points selected by a particular PCE. $\hat{\pi}$ is all join points matching a structural pattern derived from the PCE. α and β, which are related to type I and type II statistical errors, respectively, are metrics that help assess how close a structural pattern resembles a PCE in terms of selected join points.

The α and β metrics, which are related to the statistical metrics type I and type II errors, respectively, are used for measuring the similarity between a particular PCE and structural pattern (discussed in Section 3.2.1). α measures how *closely* a pattern resembles a PCE, while β measures its *completeness*.

Program elements corresponding to JPSs selected by a PCE share a *high* degree of structural commonality iff there exist structural patterns that have *small* α and β errors w.r.t. the PCE (threshold selection is discussed in Section 4.2). Conversely, elements corresponding to JPSs selected by a PCE do *not* share a high degree of commonality iff there exist patterns that have *large* α and β errors w.r.t. the PCE.

We treat a program as a set of JPSs that *may* or *may not* be under the influence of advice, which helps simplify the presentation. Also for simplification, we treat base-code as separate from the aspect, i.e., that advice cannot apply to aspect code. Doing so minimizes the types of relations that need to be considered when presenting our approach, as well as frees us from resolving **proceed** calls, which may be present in **around** advice. Future work for adding advice bodies to the analysis is discussed in [31, Section 6].

We define a PCE to be a subset of JPSs, thus eliminating the need to consider complex expression constructs. We also assume that the PCE is free of dynamic conditions, which allows us to exploit solely static information in our analysis. Our implementation conservatively relaxes this assumption (discussed in Section 4.1) so that PCEs utilizing dynamic conditions may nevertheless be used as input to our tool. The impact of this limitation is minimal [31]. Moreover, there is evidence that suggests that most PCEs do not take advantage of dynamic conditions [40].

1. (**HIGH COMMONALITY**$\wedge\neg$**SELECTED**) If a new join point is added to the base-code that *shares* a *high* degree of structural commonality (as determined by static analysis, details of which are in Section 3.2.1) with join points selected by an existing PCE and is *not* selected by the PCE, we consider the PCE to be *more* "interesting," as it *may* need to be altered to *include* the new join point.

2. (**LOW COMMONALITY**$\wedge\neg$**SELECTED**) If a new JPS is added to the base-code that does *not* share a *high* degree of structural commonality with JPSs selected by the PCE and is *not* selected by the PCE, we consider the PCE to be *less* "interesting," as it is *unlikely* that the PCE needs to *include* the new join point.

3. (**HIGH COMMONALITY**$\wedge$**SELECTED**) If a new JPS is added to the base-code that *shares* a *high* degree of structural commonality with JPSs selected by the PCE and *is* selected by the PCE, we consider the PCE to be

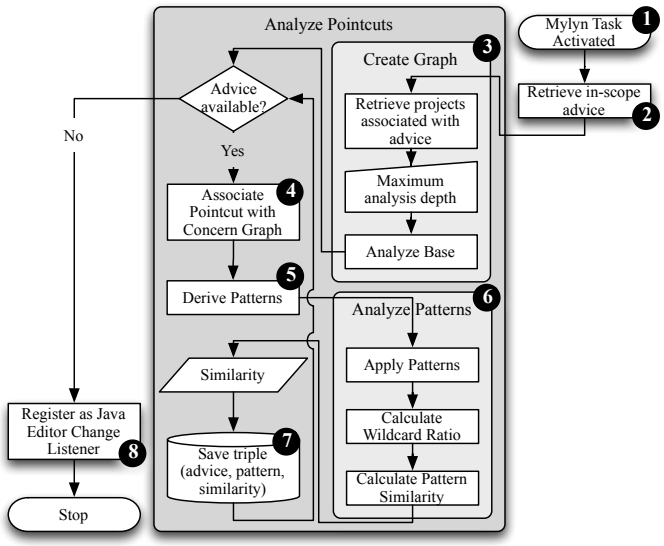


Figure 2: Flowchart of the analysis phase that starts when a task is activated.

less "interesting," as it is *unlikely* that the PCE needs to *exclude* the new join point.

4. (**LOW COMMONALITY∧SELECTED**) If a new JPS is added to the base-code that does *not* share a *high* degree of structural commonality with JPSs selected by the PCE and *is* selected by the PCE, we consider the PCE to be *more* "interesting," as it *may* need to be altered to *exclude* the new join point.

3.2. Workflow Details

Once a Mylyn task is activated, FRAGLIGHT detects broken PCEs using two phases, namely, *analysis* and *detection*.

3.2.1. Phase I: Analysis

Pointcut Analysis Scope. The analysis phase (Figure 2) is triggered when a Mylyn task is activated (step 1). At this time, a set of advice-bound PCE[10] representations is collected from the current *pointcut analysis scope* (PAS; step 2). The PAS is based on the *degrees-of-separation* concept used by [34]. It is used there (and here) to

[10]In AspectJ, it is possible to declare a (named-) PCE for which no advice is currently bound. We are considering analyzing such PCEs for future work.

limit the amount of data to be analyzed in their experimental *Active Search* feature, which provided an additional view of "interesting" elements computed by (indirect) interactions. PCEs bound to advice in this set are those that FRAGLIGHT will later consider during the detection phase (described in Section 3.2.2) when predicting broken PCEs due to added JPSs. Thus, it is these and only these PCEs that can possibly be included in our change predictions. This helps control tractability by allowing only a subset of PCEs to be analyzed. In our implementation (described in Section 4.1), and later used during our experiments (described in Sections 4.3 and 4.4), the *workspace* scope, which considers all PCEs available in all projects in a developer's workspace, is the default.

Example 2. If the aspect in Listing 2 was the only aspect in all of the projects in the workspace, the PAS would include the PCEs bound to the **after**() advice declared on line 2 and the **around**() advice declared on line 5.

Concern Graphs. In step 3, an *extended concern graph* is built from projects that include the aspects whose advice-bound PCEs are in the PAS. A concern graph is a directed multigraph depicting structural relations (e.g., calling, declarations, package containment) between program elements (e.g., types, methods, fields) [41]. We extend the graph with relations and entity types found in modern Java languages.

Example 3. Vertices for Point, Point.y, and Point.getY() would be in a graph built from Listing 1. Arcs would include Point $\xrightarrow{df}$ Point.y and Point $\xrightarrow{dm}$ Point.getY() $\xrightarrow{gf}$ Point.y, where *df*, *dm*, and *gf* refer to field declaration, method declaration, and field retrieval ("gets field") relations, respectively.

Maximum Analysis Depth. A maximum analysis depth (k) is also a parameter to control tractability. It controls the depth of the structural relations considered. In Section 4.2, we discuss our choice for the analysis depth for our experiments.

Pattern Extraction. Next, each PCE in the PAS is associated with the graph (step 4). This involves identifying portions of the graph (vertices or arcs) that are related to the JPSs selected by a PCE.

Example 4. Recall that the PCE declared on line 3 of Listing 2 selects executions of methods (and overriding methods via the + designator in Figure+) implementing the Figure interface and whose name begins with "set," etc. This PCE would be associated with the vertices representing the methods Point.setX() and Point.setTwiceX(). Graph elements (e.g., vertices) that represent such methods are "enabled" w.r.t. a PCE [31].

Algorithmically, pattern extraction works by first enumerating acyclic, finite paths of maximum length k in the graph.

Example 5. A path of length one is Point.setX() $\xrightarrow{sf}$ Point.x, where *sf* represents a field manipulation ("sets field") relation.

Next, paths that contain enabled vertices or arcs are used to construct patterns.

Example 6. The vertex Point.setX() in the path shown in Example 5 is enabled w.r.t. the PCE declared on line 3 in Listing 2.

Wild cards are then substituted for various graph elements (either vertices or arcs), with the enabled graph elements being substituted with "enabled wild cards" (step 5).

Example 7. We derive the pattern ?* $\xrightarrow{sf}$ Point.x from the PCE declared on line 3 in Listing 2 using the path depicted in Example 5, where ?* is an enabled wild card.[11] Note that the enablement is w.r.t. the PCE.

Pattern Matching. Pattern matching identifies paths with common sources and sinks as those containing enabled graph elements. Graph elements matching enabled wild cards are those whose represented JPS exhibit similar structural commonality with the JPSs selected by the PCE.

Example 8. The pattern in Example 7 would match (and only match) the paths Point .setX() $\xrightarrow{sf}$ Point.x and Point.setTwiceX() $\xrightarrow{sf}$ Point.x in Listing 1 (and only them). Notice that the enabled wild card ?* matches Point.setX() and Point.setTwiceX(), which corresponds to *all* and *only* the selected JPSs. This indicates that this pattern describes similar structural characteristics as the PCE from which it was derived. Note, though, that while the enabled wild card of the pattern Point $\xrightarrow{df}$?* also matches both Point.setX() and Point.setTwiceX(), it also matches Point.getY(), whose corresponding JPS is not selected by the PCE. This indicates that, while this pattern expresses similar structural characteristics as the PCE, it is too broad.

We next detail how patterns and PCEs are compared. Recall that patterns describe arbitrarily deep structural commonality between program elements corresponding to join points selected by a PCE.

Pattern Analysis.

[11] Patterns of greater lengths may contain wild cards that are not enabled.

$$err_\alpha(\hat{\pi}, PCE) = \begin{cases} 0 & \text{if } match(\hat{\pi}, paths(CG)) = \varnothing \\ 1 - \dfrac{|PCE \cap match(\hat{\pi}, paths(CG))|}{|match(\hat{\pi}, paths(CG))|} & \text{o.w.} \end{cases} \tag{1}$$

$$err_\beta(\hat{\pi}, PCE) = \begin{cases} 1 & \text{if } PCE = \varnothing \\ 1 - \dfrac{|PCE \cap match(\hat{\pi}, paths(CG))|}{|PCE|} & \text{o.w.} \end{cases} \tag{2}$$

$$abs(\hat{\pi}) = \begin{cases} 1 & \text{if } |\hat{\pi}| = 0 \\ \dfrac{|\mathcal{W}(\hat{\pi})|}{|\hat{\pi}|} & \text{o.w.} \end{cases} \tag{3}$$

$$sim(\hat{\pi}, PCE) = 1 - [err_\alpha(\hat{\pi}, PCE)(1 - abs(\hat{\pi})) + err_\beta(\hat{\pi}, PCE)\, abs(\hat{\pi})] \tag{4}$$

$$sel(jps, PCE) = \begin{cases} 1 & \text{if } jps \in PCE \\ 0 & \text{o.w.} \end{cases} \tag{5}$$

$$\mu(jps) = \left\{ \hat{\pi} \;\middle|\; jps \in match(\hat{\pi}, paths(CG')) \right\} \tag{6}$$

$$\delta(PCE) = \left\{ \hat{\pi} \;\middle|\; \hat{\pi} \text{ was derived from } PCE \right\} \tag{7}$$

$$chconf(jps, PCE) = \begin{cases} sel(jps, PCE) & \text{if } \mu(jps) \cap \delta(PCE) = \varnothing \\ \dfrac{1}{|\mu(jps) \cap \delta(PCE)|} \displaystyle\sum_{\hat{\pi} \in \mu(jps) \cap \delta(PCE)} |sel(jps, PCE) - sim(\hat{\pi}, PCE)| & \text{o.w.} \end{cases} \tag{8}$$

Figure 3: PCE change confidence equation. The first four are derived from [31]. CG is the concern graph built from the original base-code when the Mylyn task is activated. CG' is the revised graph, which includes information pertaining to the new join point shadow *jps*. err_α is the type I error ratio. err_β is the type II error ratio. $|\mathcal{W}(\hat{\pi})|$ is the number of wild cards in the pattern $\hat{\pi}$. *abs* is the pattern abstractness, i.e., the ratio of wild card to concrete elements. $sim(\hat{\pi}, PCE)$ represents how well the pattern $\hat{\pi}$ resembles the PCE PCE. The characteristic function $sel(jps, PCE)$ is 1 if the JPS *jps* is *selected by* the PCE PCE. $\mu(jps)$ is the set of all patterns that, when applied to the new version of the base code CG', produce *jps* as a result. $\delta(PCE)$ is the set of all patterns derived from PCE. $chconf(jps, PCE)$ (the "change confidence") is the *confidence* we have that PCE will need to be changed (i.e., it "breaks") given that the JPS *jps* was added to the base-code. The closer *chconf* is to 1, the more likely PCE will need to be updated because of adding *jps*. Conversely, the closer *chconf* is to 0, the less likely PCE will need to be updated because of adding *jps*.

Step 6 is responsible for comparing the derived patterns with the PCE (as demonstrated above) and producing a pattern *similarity* metric, which quantifies how closely the pattern resembles a PCE in terms of structural properties related to selected JPSs. The closer a pattern's similarity is to 1 (its range is in $[0, 1]$), the more closely the pattern matches similar structural commonality as that of the PCE. The equation to calculate the pattern-PCE similarity[12] is depicted in equation (4) of Figure 3.

Details of the pattern similarity metric are as follows. *CG* refers to the extended concern graph built from the original base-code when the Mylyn task is activated in step 3. In our motivating example, this graph would represent the code in Listing 1. Next, we define a function $match(\hat{\pi}, \Pi)$, where $\hat{\pi}$ ranges over the set of patterns and Π the power set of paths in *CG*. This function, given a pattern and a set of paths, matches the pattern against the paths, resulting in a set of JPSs. These are the JPSs whose corresponding program elements exhibit the structural commonality represented by the pattern.

Equations (1), (2), and (3) are combined in the similarity calculation to measure patterns on three dimensions. Equation (1) is the err_α error rate attribute (cf. α discussed in Section 3.1), which is akin to the ratio of the number of JPSs selected by both the PCE and the pattern when matched against finite, acyclic paths in the graph $paths(CG)$ to the number of JPSs solely selected by the pattern ($|PCE|$ refers to the number of JPSs selected by *PCE*). It is subtracted from 1 to create an error ratio in the statistical sense. It quantifies the pattern's ability in matching *solely* the JPSs within the PCE; the closer the err_α rate is to 0 the more likely the JPSs matched by the pattern are also ones within the PCE. If $\hat{\pi}$ does not match any JPSs, the err_α is 0 as it is vacuously precise.

Example 9. The pattern depicted in Example 7 would have a *small* (in fact, 0) err_α w.r.t. the PCE declared on line 3 of Listing 2, as both express exactly the same methods, namely, Point.setX() and Point.setTwiceX(). On the other hand, the pattern Point $\xrightarrow{dm}$?* would have a *larger* err_α w.r.t. the PCE declared on line 6 as the executions of Point.setX() and Point.setTwiceX() would be matched by the pattern but not selected by the PCE. Particularly, err_α here would be $\frac{2}{3}$ because, of the three method **execution**s matched by the pattern, only one of them is also selected by the PCE ($1 - \frac{1}{3}$).

[12]The similarity metric is equivalent to the confidence equation in [31]. *Similarity* is used here as *confidence* is reserved for the PCE *change confidence* metric discussed later. Moreover, the maximum analysis depth (path length) parameter k is made implicit for presentation purposes.

Equation (2) is the err_β error rate attribute, which is akin to the ratio of the number of JPSs selected by both the PCE and the pattern when applied to paths in the graph to the number of JPSs selected solely by the PCE. Similar to err_α, the quantity is subtracted from 1 and its range is in $[0, 1]$. It quantifies the pattern's ability in matching *all* of the JPSs selected by the PCE; the closer the err_β rate is to 0 the more likely the pattern is to match all the JPSs selected by the PCE. If there are no JPSs selected by the PCE, the err_β is vacuously 1 ($\forall\hat{\pi}\forall CG[\varnothing \subseteq match(\hat{\pi}, paths(CG))]$).

Example 10. The pattern shown in Example 7 would have a *small* (in fact, 0) err_β w.r.t. the PCE declared on line 3 of Listing 2, as the pattern matches *all* of the methods selected by the PCE (i.e., the pattern "covers" the PCE). However, the same pattern would have a *large* (in fact, 1) err_β w.r.t. the PCE declared on line 6 of Listing 2, as *none* of the method executions matched by the pattern are selected by the PCE (i.e., it does not cover the PCE).

Finally, equation (3) is the pattern abstractness (abbreviated *abs*), i.e., the ratio of wild card to concrete elements. $\mathcal{W}(\hat{\pi})$ projects the wild cards from a pattern $\hat{\pi}$, with $|\mathcal{W}(\hat{\pi})|$ being the number of wild cards in the pattern $\hat{\pi}$ and $|\hat{\pi}|$ being the total number of graph elements. An empty pattern has no concrete elements, thus, it has an *abs* of 1. For instance, the pattern in Example 7 has an *abs* of $\frac{1}{3}$.

We use *abs* because patterns containing many wild cards are more likely to match a greater number of concrete graph elements and vice versa. Thus, we combine the err_α and err_β rates by use of a weighted mean weighted by *abs* in equation (4). The reason is that a pattern that is very abstract is less likely to match JPSs that are *only* selected by a PCE. On the other hand, a pattern that is less abstract is less likely to match all JPSs selected by a PCE [31].

Example 11. Let $\hat{\pi}$ be the pattern from Example 7, *PCE* be the PCE declared on line 3 of Listing 2, and *CG* be the graph representing the base-code in Listing 1. Then, $sim(\hat{\pi}, PCE) = 1 - [(0)(\frac{2}{3}) + (0)(\frac{1}{3})] = 1$. Let $\hat{\pi}$ be $\mathsf{Point} \xrightarrow{dm} ?^*$ and *PCE* be the PCE declared on line 6. Then, $sim(\hat{\pi}, PCE) = 1 - [(\frac{2}{3})(\frac{2}{3}) + (0)(\frac{1}{3})] = \frac{5}{9}$.

Once the pattern similarity has been calculated, triples corresponding to an analyzed advice, a pattern derived using its bound PCE, and the pattern's similarity to the PCE are stored in memory (step 7) for later use in the (next) detection phase. When all PCEs have been processed, FRAGLIGHT is registered as a *Java Editor Change Listener* [42] (step 8). In this way, it becomes an "observer" of the editing pane where the base-code developer writes code. This allows FRAGLIGHT to observe keystrokes entered by the developer and detect when a new JPS is added; we explain

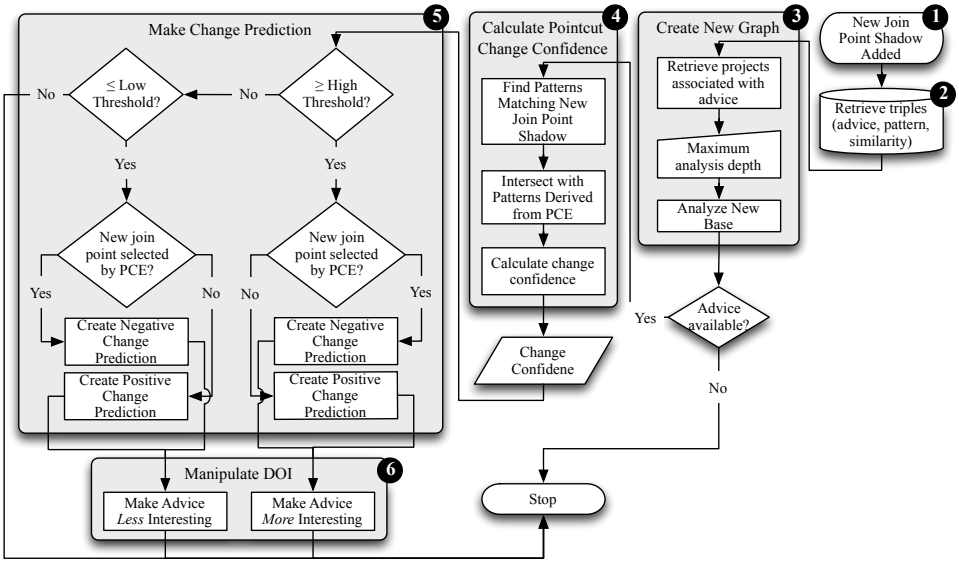


Figure 4: Flowchart depicting the detection phase that commences when a new join point is added.

this in more detail in the following section. Once a Mylyn task is deactivated, the tool is de-registered as a java editor change listener.

3.2.2. Phase II: Detection

In the detection phase (Figure 4), FRAGLIGHT determines new JPSs when keystrokes are entered by the developer in the IDE (step 1). For method execution JPSs, it finds new method declarations using Eclipse [43], which are the lowest level granularity whose addition information is available by this framework. FRAGLIGHT then includes its own code for JPSs residing within method bodies, e.g., method calls, adapting an AST differencing algorithm [44]. The new JPSs that FRAGLIGHT would detect are shown in Example 1.

Triples related to analyzed advice (PCEs), patterns, and similarity (calculated in the analysis phase) are retrieved in step 2. Then, the graph (CG) is augmented with information pertaining to the new base-code version using projects associated with the retrieved advice (resulting in CG', step 3).

Example 12. Adding the move() method in Listing 3 would result in new paths, e.g., Point $\xrightarrow{dm}$ move(), move() $\xrightarrow{sf}$ Point.x, move() $\xrightarrow{sf}$ Point.y, being added to CG, producing CG'.

Next, for each retrieved advice, its bound PCE *change confidence* (defined in equation (8)) value is calculated (step 4). First, we define a characteristic function

sel in equation (5) s.t. $sel(jps, PCE) = 1$ if *jps* is selected by *PCE* and 0 otherwise. Recall that we treat a program as consisting of a set of JPSs that may or may not be currently selected by a PCE and treat a PCE as selecting a subset of these JPSs. As such, a *jps* is selected by *PCE* iff $jps \in PCE$.

Example 13. Let *jps* = **execution(void** Point.move(**double**,**double**)) and *PCE* be the PCE declared on line 3 of Listing 2. Then, we have that $sel(jps, PCE) = 0$ because, although move is a method of a class implementing Figure, its name does not begin with "set". Let *jps* = **execution(void** Point.setX(**double**)). Then, $sel(jps, PCE) = 1$.

In equation (6), $\mu(jps)$ is the set of all patterns that match *jps* when applied to the new base-code version CG'.

Example 14. Let *jps* = **execution(void** Point.move(**double**,**double**)), $k = 1$, and CG' be the graph representing the combined base-code of Listings 1 and 3. Then, $\mu(jps) = \{?^* \xrightarrow{sf} \mathsf{Point.x}, ?^* \xrightarrow{sf} \mathsf{Point.y}, \mathsf{Point} \xrightarrow{dm} ?^* \}$.

In equation (7), $\delta(PCE)$ is all patterns derived from *PCE* (obtained from step 2 of Figure 4).

Example 15. Let *PCE* be the PCE declared on line 3 of Listing 2. Then, $\delta(PCE) = \{?^* \xrightarrow{sf} \mathsf{Point.x}, \mathsf{Point} \xrightarrow{dm} ?^* \}$. Let *PCE* be the PCE declared on line 6. Then, $\delta(PCE) = \{?^* \xrightarrow{gf} \mathsf{Point.y}, \mathsf{Point} \xrightarrow{dm} ?^* \}$.

Finally, Equation (8) depicts the PCE change confidence equation, which produces a real number in $[0, 1]$ that corresponds to the *confidence* we have that *PCE* will need to be *changed* (i.e., it breaks) due to adding *jps* to the base-code. The closer the value is to 1, the more likely the PCE breaks because of the new JPS and vice-versa.

We now discuss the individual cases within equation (8). The case in which $\mu(jps) \cap \delta(PCE)$ is non-empty implies that there is at least one pattern s.t. it is derived from *PCE* and it matches *jps*, which is part of the new base-code. We consider the *similarity* of all such patterns to *PCE*. If a pattern is very similar to the PCE in terms of matching and selected JPSs, respectively, and *jps* is not selected by the PCE, i.e., $sel(jps, PCE) = 0$, then we are very confident that *PCE* has broken as a result of adding *jps*. In this case, we have that $|sel(jps, PCE) - sim(\hat{\pi}, PCE)|$ will be close to 1. This situation corresponds to the top left (1) Venn diagram in Figure 1. Each of the other Venn diagrams corresponds to situations where the limits of *sel* and *sim* go to 0 and 1, respectively. The equation is then the average of the values for all patterns meeting the earlier stated criteria. If no patterns meet this criterion, i.e., $\mu(jps) \cap \delta(PCE) = \varnothing$, then the change confidence is simply whether

or not the JPS is selected by the PCE, i.e., $sel(jps, PCE)$. This is because there are no patterns derived from the PCE that also match *jps*.

The reasoning behind equation (8) in Figure 3 is as follows. When $\mu(jps) \cap \delta(PCE) = \varnothing$, none of the patterns derived from PCE, i.e., $\delta(PCE)$, matches *jps* as a result of applying them to CG'. In other words, *jps* shares *no* structural commonality with JPSs selected by PCE. Being that our hypothesis is that JPSs selected by a PCE typically share significant structural commonality, and this JPS shares *no* structural commonality with such JPSs, we suggest that *jps not* be selected by PCE. Then, the confidence we have in PCE breaking as a result of adding *jps* is just $sel(jps, PCE)$, i.e., 1 if *jps* is selected by PCE and 0 otherwise. In contrast, when $\mu(jps) \cap \delta(PCE) \neq \varnothing$, there exists a pattern derived from PCE that matches *jps* as a result of applying it to the new base-code. Here, we average the *chconf* for all such patterns.

Example 16. Let *jps* = **execution(void** Point.move(**double**,**double**)), PCE be the PCE declared on line 3 of Listing 2, $k = 1$, and CG' be the graph representing the combined base-code of Listings 1 and 3. Per Examples 14 and 15, we have that

$$|\mu(jps) \cap \delta(PCE)| = |\{?^* \xrightarrow{sf} \mathsf{Point.x}, \mathsf{Point} \xrightarrow{dm} ?^*\}| = 2$$

As such, we have that $chconf(jps, PCE)$

$$= \frac{1}{2}\Big(|sel(jps, PCE) - sim(?^* \xrightarrow{sf} \mathsf{Point.x}, PCE)| + |sel(jps, PCE) - sim(\mathsf{Point} \xrightarrow{dm} ?^*, PCE)|\Big)$$

$$= \frac{1}{2}\left(|0 - 1| + |0 - \frac{7}{9}|\right) = \frac{8}{9} \text{ (per Examples 11 and 13)}$$

Let PCE be the PCE declared on line 6. Then,

$$|\mu(jps) \cap \delta(PCE)| = |\{\mathsf{Point} \xrightarrow{dm} ?^*\}| = 1$$

As such, we have that $chconf(jps, PCE)$

$$= |sel(jps, PCE) - sim(\mathsf{Point} \xrightarrow{dm} ?^*, PCE)|$$

$$= |0 - \frac{5}{9}| = \frac{5}{9} \text{ (per Examples 11 and 13)}$$

Notice that the *chconf* of the *broken* PCE (line 3) is *greater* than the *chconf* of the *unbroken* PCE (line 6).

3.2.3. PCE Change Prediction

A PCE *change prediction* is created for PCEs with change confidences either below a *low* or above a *high* threshold (step 5). As a convenience, we add additional information regarding the prediction depending on whether the newly added JPS is currently selected by the corresponding PCE. It is meant to guide the developer in determining not only that a particular PCE is broken but also in how a broken PCE should be fixed, i.e., whether the new JPS should be removed from (i.e., a *negative* change prediction) or added to (i.e., a *positive* change prediction) the PCE.

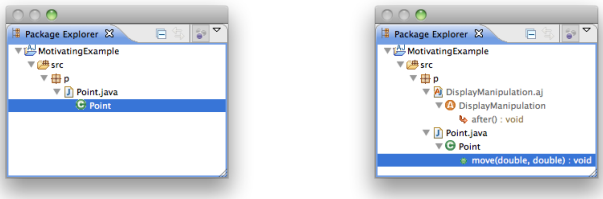
3.2.4. Mylyn DOI Model Manipulation

The Mylyn Eclipse IDE plug-in maintains focused contexts of entities relevant to a particular task using a Degree of Interest (DOI) model. A context is comprised of the relevant elements (e.g., classes, methods, fields), along with information pertaining to how *interesting* the elements are to the related task. The more a developer interacts with an element (e.g., navigates to a file, edits a file) when working on a task, the more interesting the element is deemed to be, and vice-versa. Mylyn then alters the behavior of the Eclipse workbench such that only interesting elements are displayed throughout the various workbench views.

In Mylyn, elements may also become interesting implicitly, e.g., a package may become interesting if a class within the package is edited. FRAGLIGHT implicitly manipulates the Mylyn DOI model (step 6) using the low and high confidence thresholds. If the PCE change confidence falls in the *low* confidence interval, the PCE is made *less* "interesting" in the DOI model, moving the developer's attention *away* from the PCE so that they may focus on the base-code. Conversely, if the change confidence falls in the *high* interval, the PCE is made *more* "interesting," bringing the developer's attention *towards* the PCE, so that they may focus on PCEs that may have broken as a result of their newly added base-code.

Example 17. Due to the small size of our example, let the low *chconf* threshold be 0.6 and the high be 0.8. The scenario described in Example 16 results in a positive change prediction for the PCE declared on line 3 of Listing 2 as its *chconf* is above the high threshold, thereby increasing the PCE's DOI value. Conversely, the PCE declared on line 6 has a *chconf* below the low threshold, which results in a negative change prediction and a decrease in its DOI value. As such, the broken PCE receives a higher DOI value than the unbroken one.

Figure 5(a) shows the Eclipse package explorer, whose contents are filtered by Mylyn, prior to adding the move() method to the Point class. Notice that no PCEs initially appear in this view. Figure 5(b) portrays the same package explorer but after adding the move() method, which breaks the PCE declared on line 3 of Listing 2



(a) Before adding move(). (b) After adding move().

Figure 5: Screenshot of the Eclipse "Package Explorer" before and after adding the move() method. Note that the developer does not interact with the DisplayManipulation aspect in any way. The sole reason for its appearance is due to FRAGLIGHT's programmatic DOI manipulation.

but not the PCE declared on line 6. FRAGLIGHT programmatically manipulates the DOI model, causing the broken associated advice to appear but not the unbroken advice. Other views, i.e., the structure explorer, displaying the advice will be similarly affected.

4. Experimental Evaluation

In this section, we detail an evaluation of our approach that assesses its accuracy in detecting broken PCEs.

4.1. Implementation

FRAGLIGHT is implemented as a relation provider extension to the standard Mylyn Eclipse plug-in. The extended concern graph was constructed using the JayFX fact extractor [41], which we extended for use with modern Java languages and AspectJ (with the latter being as part of our previous work [31]). JayFX generates "facts," using class hierarchical analysis (CHA) [45], pertaining to structural properties and relationships between program elements, e.g., field accesses, method calls, in a particular project. Its lightweight representation of program elements makes for an efficient analysis. Source code and transitively referenced libraries (possibly in binary format) are analyzed during graph building.

The AJDT (`http://www.eclipse.org/ajdt`) compiler was leveraged to conservatively (explained next) associate the graph with a PCE. For a given PCE, the AJDT compiler produces the Java program elements, e.g., method declarations, method calls, field sets, correlated with selected JPSs. Both pattern extraction and pattern-path matching were implemented via the Drools Rules Engine (`http://www.drools.org`), which uses a modified RETE algorithm [46]. Drools

provides a natural query language and an efficient solution to the many-to-many matching problem. A prototype implementation of FRAGLIGHT is publicly available (`http://github.com/khatchad/fraglight`).

4.2. Study Configuration

To assess the accuracy of our approach in detecting broken (and unbroken) PCEs, we examined the final DOI values of PCEs (as manipulated by only FRAGLIGHT) after replaying a series of base-code changes from software version histories. Although this approach may seem course-grained, particularly since our tool works on fine-grained changes, we use major release points from software version histories so that we may assume that PCEs in those versions are written correctly. This allows us to build an oracle for which to compare the results of our tool. Possible drawbacks are discussed in Section 4.5.

Elements with a higher DOI value will be more prominent in the IDE and will thus be more noticeable by developers and vice-versa. An assessment using traditional precision and recall metrics are not directly applicable in this situation due to the DOI model [32, 34]. That is, the DOI model is a scale; adding weight to one element removes it from another. It is very much a sorting mechanism that strives to display the most relevant IDE UI elements to the developer for a specific task. In this way, comparing the ratio of DOI values between broken and unbroken PCEs suffices as an effective assessment.

With the initial DOI value of each PCE at 0, we say that a successful DOI manipulation is one where broken PCEs had a higher DOI value than those that did not break. In this case, broken PCEs are brought to the developer's attention, whereas PCEs that did not break remain in the background. Note that it is important, during the experiments, not to manually manipulate the DOI, e.g., by manually clicking on any of the IDE elements so that we can be sure that FRAGLIGHTś predictions are solely responsible for these values. We consider possible drawbacks of this approach in Section 4.5.

For this experiment, we set $k = 1$ (see Section 3.2.1), which keeps the tool run time short so that predictions can be made as quickly as possible since the analysis runs while the developer is typing. Moreover, we set the low and high confidence thresholds parameters to 0.15 and 0.55, respectively, meaning that PCEs assigned a *chconf* ≤ 0.15 resulted in a decreased DOI, while ones of ≥ 0.55 resulted in an increase. We empirically found that these thresholds worked the best with our corpus. In the future, we plan to more thoroughly assess trade-offs between analysis depth and prediction time, as well as optimal threshold values.

subject	vers.	LOC	aPCE	at (s)	JPS	pt (s)	bPCE	uPCE	bDOI	σ_{bDOI}	uDOI	σ_{uDOI}	p-value
HealthWatcher	8	47537	217	47.20	2648	1.1e4	6	29	1.17	0.98	0.21	0.77	0.031473091
MobilePhoto	6	8331	196	11.18	3063	3.5e3	29	59	2.21	2.54	1.32	2.26	0.058945232
Totals:	14	55868	413	58.39	5711	1.5e4	35	88	2.03[a]	2.37[a]	0.95[a]	1.97[a]	0.01059113

[a]Arithmetic mean

Table 1: Experimental results. *vers.* is the number of versions analyzed, *LOC* is the total number of non-blank, non-commented lines of code, *aPCE* is the total number of PCEs analyzed throughout the versions, *at (s)* is the total analysis time in secs, *JPS* is the total number of JPSs added between versions, *pt (s)* is the total prediction time in secs, *bPCE* and *uPCE* are the total number of broken and unbroken PCEs between versions, respectively, *bDOI* and *uDOI* are the average final DOI value of broken and unbroken PCEs, respectively, with the σ columns containing the corresponding standard deviations. Column *p-value* represents the p-value from a Welch's t-test on our dataset, indicating a significant difference in final DOI value between the two kinds of PCEs (i.e., broken and unbroken).

Table 1 includes our subjects along with associated number of discrete releases (column *vers.*) analyzed (the initial version is *not* included in this column), total non-blank, non-commented lines of code (counted using SLOCCount; `http://www.dwheeler.com/sloccount`), which excludes code contained within aspect files, between all versions (column *LOC*), ranging from an average of $\approx$ 1.4K per version for MobileMedia and $\approx$ 6K for HealthWatcher, and total number of analyzed (advice-bound) PCEs throughout versions (column *aPCE*).

To ensure that a certain level of quality was maintained, we purposefully selected subjects that have been used previously in the literature [47, 48]. This ensures that the subjects have achieved a particular level of acceptance within the community. Although only two subject projects were used, fourteen total versions were analyzed, yielding a reasonably sized corpus.

HealthWatcher is a web-based application that provides various medical-related support to patients. MobileMedia is a software product line consisting of applications that manipulate photo, music, and video on mobile devices. Subject source code, descriptions, and references can be found on our website (`http://openlab.citytech.cuny.edu/pcp`). The authors were not involved in the development of any of the subject applications.

Column *at (s)* depicts the total PCE analysis time in secs for all versions. Analysis occurs when the developer activates a Mylyn task (normally at the start of working on a particular bug or feature). Then, all PCEs in the PAS are analyzed. For each version, the analysis was repeated three times, with the results of each averaged, using a 2.83 GHz Intel machine. The JVM heap size was 5 GB. The average was $\approx$ 1.05 secs per KLOC and $\approx$ 0.14 secs per PCE, which indicates that the analysis time is practical for even large applications.

Column *JPS* is the total number of JPSs added between subject versions. These are the JPSs used as input, some of which broke PCEs and others that did not. Since we collected PCE statistics after inputting all JPSs added between versions to our tool, it was not important to identify precisely which JPSs caused particular PCEs to break. Instead, classifying which PCEs broke and which did not was sufficient.

As noted in Section 3, the output of our tool is a PCE change prediction, which, in turn, manipulates the Mylyn DOI. All of this is done in the background as the developer is working. We should note also that the predictions occur in a separate thread, which fortunately does not interrupt the developer's workflow. However, having short prediction times (i.e., the amount of time needed for FRAGLIGHT to generate a PCE change prediction) is advantageous so that broken PCEs are brought to the developer's attention as early as possible. Column *pt (s)* portrays the total prediction time in secs during our experiment, which averaged $\approx$ 2.61 per added JPS.

This indicates that the developer would see programmatic changes in the DOI made by FRAGLIGHT on average ≈ 2.61 secs after adding a new JPS to the base-code, which is practical. The developer is not blocked as the update happens concurrently. The remaining columns will be discussed shortly.

The order in which JPSs were used as input to our tool is insignificant. This may be unintuitive as applying patterns to different base-codes is likely to match different JPSs. However, the only base-code of concern for matching the patterns is the JPS being added. As such, the order in which JPSs are added to the old base-code version to obtain the new base-code version is irrelevant as each JPS is considered in isolation.

Columns *bPCE* and *uPCE* are the total number of broken and unbroken PCEs between versions, respectively. For this, we use the conditions for a PCE to be considered broken between subsequent base-code versions from [31]. We say that a PCE in version v_i *broke* in version v_j where $i < j$ iff both of the following conditions hold:

1. the textual representation of the PCE in v_i differs from its textual representation in v_j,
2. the JPSs selected by the PCE in v_j differs from the JPSs selected by the textual representation of the PCE in v_i in v_j.

Criterion 1 asserts that the PCE was rewritten between versions, i.e., they textually differ. Criterion 2 excludes situations where the PCE selects the same JPSs between versions.

PCEs that meet these criteria are those that required textual modification to allow the PCE to continue to capture intended join points. We discuss possible drawbacks for using these criteria to identify broken PCEs in version history in Section 4.5.

4.3. Quantitative Analysis

After simulating the addition of JPSs between versions of our subjects, we then collected the resulting PCE DOI values. The hope is that broken PCEs resulted in a higher DOI value than that of unbroken PCEs. In this case, broken PCEs would appear more prominently in the IDE than unbroken PCEs, so that developers can direct their attention to the problem early. Columns *bDOI* and *uDOI* depict the average final DOI value of broken and unbroken PCEs, respectively, while σ_{bDOI} and σ_{uDOI} portray the corresponding standard deviations. These columns show the average final PCE DOI values after adding all the new JPSs between versions v_i and

v_{i+1} to v_i for all $i = 1 \ldots k-1$ where k is the number of subject versions. As we can see from column *p-value*, the averages of DOI values of broken and unbroken pointcuts are significantly different, i.e., they are under the 0.1 significance level.

From Table 1, the average DOI value of PCEs that *actually* broke was, on average, 2.14 times greater than the average DOI value of PCEs that *did not* break. Also note that the average corresponding standard deviations across subjects of broken and unbroken PCEs are 2.37 and 1.97, respectively. In Section 4.4, we discuss several outliers in these results and possible reasons for the distributions.

Recall that the resulting DOI values are completely and only due to FRAGLIGHT's manipulation. These results indicate that FRAGLIGHT is promising in bringing broken PCEs to the developers' attention while hiding unbroken PCEs, all while they are typing. Particularly, using our approach results in broken PCEs being times more *prominently* displayed in the IDE than unbroken PCEs. Moreover, FRAGLIGHT presents its results to the developer in a familiar way using existing, well-integrated IDE mechanisms (i.e., Mylyn). Because of Mylyn, FRAGLIGHT's results are propagated throughout all UI elements where PCEs are visible, making the results consistent among views.

Also recall that FRAGLIGHT provides base-code developers with feedback following a series of related, fine-grained base-code changes that may have broken PCEs. The evaluation presented here, however, considers DOI changes after a series of input JPSs. More specifically, we portray the average final DOI value of each PCE between versions. The basis for this is as follows. FRAGLIGHT may manipulate, depending on the threshold settings, a PCE's DOI after each individual JPS addition. The exact amount by which the interestingness level is changed is solely dictated by the Mylyn framework. In practice, we found this amount to be quite small. Thus, the more JPSs that are added that break a PCE, the more interesting the PCE will become in the Mylyn context, and vice-versa. To more accurately evaluate our approach, we used fine-grained base-code changes, as those are used as input to our tool, but propagated the results amongst a series of changes to assess whether broken PCEs would eventually surface and unbroken PCEs recess in the IDE. In the future, we plan to investigate varying the PCE interestingness level amount during manipulation, perhaps making it proportional to the degree of change confidence.

4.4. Qualitative Analysis

We now analyze several situations where our tool performed as expected and vice-versa. For succinctness, we draw examples from only the HealthWatcher subject.

We begin with a scenario where our tool assigned a *high* DOI to a PCE that *broke* between versions. The aspect synchronizes methods in "record" types using

a concurrency manager. The changed PCE in version 9 broke between versions 8 and 9 due to adding new record types (representing diseases and symptoms), which resulted in the PCE selecting two additional join points. Considering these new join points relative to the PCE in version 8, adding these new join points caused FRAGLIGHT to produce 2 predictions, averaging a *chconf* = 0.67. One structural pattern that was used was one matching accesses to a field. The JPSs added in the subsequent version were methods that also accessed this field, and being that they were not selected by the original PCE, FRAGLIGHT increased the PCE's DOI value to 2. This scenario corresponds to situation 1 in Figure 1. The DOI value was not higher because there is another method in an unrelated class that also accesses this field but is not selected by the PCE.

We now examine how other approaches would have worked in the above situation where FRAGLIGHT was successful. The AJDT would not display the broken PCE as matching (via Eclipse markers) in the base-code since it only displays currently matching information. As the approach of [28] augments the AJDT with *almost matching* join point information, it is conceivable that this approach may insert markers to the broken PCE at the appropriate places in the base-code, i.e., where the new join points were added. However, [28] does not use substring matching as part of their PCE relaxers. Although the new record type names contain a common segment (i.e., "Record"), this approach would also not be able to discover the broken PCE. The approach of [7] would likewise not be able to detect the broken PCE as the PCE of version 8 does not exhibit any difference in advice application between the two versions (i.e., it selects the same join points in both versions).

We now discuss an instance where our tool assigned a *low* DOI to a PCE that *broke* between versions. Changes made in versions 1 to 2 involved introducing the Command design pattern [49] to replace the individual servlets that implemented each of the operations provided by HealthWatcher. Consequently, a PCE in an aspect responsible for computation distribution broke. To rectify the problem, the PCE was rewritten to no longer select join points contained in classes derived from a particular servlet but instead to select join points contained in classes derived from a servlet following the Command design pattern. No base-code changes were made other than the renaming of these classes to reflect that they are related to the Command design pattern. The change was modeled as 19 JPS removals and 19 JPS additions between the two version. Unfortunately, the final DOI value for this PCE was 0 as FRAGLIGHT produced no predictions for this PCE. The reason was that all structural patterns that were derived from the PCE in version 1 were invalidated by version 2. That is, the program elements referred to in the structural patterns derived from the PCE in the first version were no longer present in the second. As

such, applying the structural patterns from version 1 produced no matches in version 2. Thus, no predictions were made, which suggests that our approach may not be effective in situations involving widespread, atomic refactorings.

The approach of [7] would detect the broken PCE in this situation as there is a difference in advice application between the two versions. Still, however, the developer would be required to manually verify that the difference in advice application is not desirable (there are situations, in fact, where it is desirable, e.g., when a PCE is robust to base-code changes and new join points are correctly selected).

It is also conceivable that the approach of [29] would be able to uncover this broken PCE as their approach is heavily tied to automated refactorings. However, not all refactoring (especially to design patterns) can be fully automated in an atomic fashion. We predict that our tool would perform well in situations where the changes involve several intermediate steps, which would provide FRAGLIGHT the opportunity to match more existing structural patterns against old base-code.

Next, we turn to an instance where our tool assigned a *low* DOI to a PCE that *did not* break between analyzed versions, namely, a PCE in a synchronization aspect. Its final DOI value following the experiment was 0.[13] As previously discussed, HealthWatcher was refactored to use the Command design pattern between versions 1 and 2. However, this PCE selected join points not related to this refactoring and thus did not break. Four predictions were made between these versions, resulting in an average $chconf = 0.11$. The derived structural patterns did not exhibit strong commonality with that of the PCE as they expressed calls to such common methods as `String.equals()`. Moreover, the structural patterns did not match the new JPSs. This resulted in a low PCE DOI value and corresponds to situation 2 in Figure 1.

Since the refactorings did not involve this particular PCE, the AJDT would not place editor markers linked to it near the base-code where the refactoring occurred. However, this unbroken PCE would still be present in other views, perhaps deterring the developer from focusing on the broken PCE described earlier. FRAGLIGHT, on the other hand, would programmatically manipulate the Mylyn DOI to reduce the focus on the unbroken PCE.

Finally, we detail a scenario where our tool assigned a *high* DOI value to a PCE that *did not* break. One such instance occurred with a PCE in an aspect responsible for catching exceptions raised inside Observer design pattern [49] implementations and displaying exception details in a web page. FRAGLIGHT produced 17 change predictions between all versions for the PCE, resulting in an average

[13] Mylyn does not currently allow for negative DOI values. As such, 0 is the lowest allowable DOI value of any element.

chconf = 0.48. The final DOI value for this PCE was 3. The PCE was **execution(void** Update∗Data.executeCommand(..)), meaning that executeCommand() methods declared in classes whose name starts with Update and ends with Data are selected (part of the Command design pattern mentioned earlier). One of the extracted structural patterns matched method executions of methods that textually included calls to CommandRequest.isAuthorized(). An added JPS in version 8, namely, the execution of the method UpdateSymptomSearch.executeCommand() (also part of the Command design pattern), includes a call to CommandRequest.isAuthorized(), as such, it matches the structural pattern. However, the PCE correctly selects base-code pertaining to the *Observer* design pattern (i.e., the *Data* is "observed") and not that of *Search*es. Since the CommandRequest.isAuthorized() is called from within many of the *Data* classes, and the new JPS matched the structural pattern (corresponding to situation 1 in Figure 1), this misled our tool to suggest that the PCE had broken.

Comparing to other approaches, the AJDT would not have created an editor marker consisting of a link to the unbroken PCE near the above mentioned newly added JPS in version 8 since the PCE does not match the new join point. Likewise, the approach of [7] would not have detected advice application differences for the unbroken PCE. The approach of [28], however, via their *DeclaringTypeRelaxer* and *NameRelaxer*, would exhibit a similar problem to that of FRAGLIGHT in this instance. This is due to the textual similarity between the selected method names (i.e., they are both named executeCommand).

4.5. Threats to Validity

Several threats may diminish our evaluation results. We discuss here how their effects have been minimized. Our evaluation aimed to simulate FRAGLIGHT's performance in a real-world setting. We drew data from multiple versions of two projects, which may not be representative of AO projects at large. However, these subjects have been extensively studied previously in the literature. Moreover, they comprise publicly available open source projects, which are contributed to by a number of developers. Although only two projects were used, they have a rich release history, constituting fourteen versions with large deltas (a total of 5,711 added JPSs).

Although a user study would be useful, we chose a software evolution simulation using version histories for several reasons. Firstly, user studies have a number of barriers [50]. Secondly, we desired to isolate the DOI manipulation that was due to our tool and not by other UI interactions performed by the developer. In this way, we can ensure that we accurately assess FRAGLIGHT's change predictions. Thirdly, Mylyn has been proven to be effective via previous user studies [34, 32], and our approach simply enhances Mylyn's prediction capabilities. The aforementioned studies

indicate that Mylyn is a successful vehicle for conveying predictions to developers. Lastly, using version histories provides a noninvasive, unbiased way to assess our approach. Nevertheless, we are considering a user study for future work to enhance the results presented here.

We assumed that all PCEs are correctly written between version deltas, which correspond to major subject release points. Our assumption is that, prior to a major release, all PCEs select and only select intended join points. This is essential in determining which PCEs broke and in which versions. Moreover, we assumed that broken PCEs were fixed by rewriting the PCE. Yet, there are other ways to "fix" a broken PCE, namely, by changing the *base-code* to conform to the PCE. For example, to fix the broken PCE portrayed in Section 2, we could change the name of the move method to setBothXandY.

It would be difficult to use base-code conformance as a reliable means to determine broken PCEs as there are many reasons that base-code can change, including fixing a broken PCE. However, it is reasonable to assume that the only reason PCEs change is because they are broken.

When assessing the changes of DOI in our experiments, we began with the DOI flat, i.e., 0, and then fed a series of added JPSs to the tool to obtain the subsequent version. No other factors affected the DOI other than the programmatic manipulation performed by FRAGLIGHT, which allowed us to focus on the quality of its predictions. However, in a real-world setting, the DOI may be affected by other events that occur within the IDE, such as developer clicks and navigation. As such, more investigation may be necessary to assess the effectiveness of FRAGLIGHT's programmatic DOI manipulation in combination with other events while the developer is typing, which we plan for future work. Furthermore, there is a possibility that the visualization Mylyn offers is too subtle for base-code developers to take action when breaking PCEs. However, the evaluation presented by [34] successfully shows that Mylyn's visualizations help developers to be more productive, which suggests that developers indeed pay attention to the views focused by Mylyn. Even so, thoroughly investigating this subtlety, especially w.r.t. AOP development, perhaps through a user study, would be an interesting avenue of future work.

5. Related Work

PCDiff [7] is an approach and corresponding tool that calculates differences in advice application following base-code changes. Specifically, given a set of base-code changes, it computes all advice that additionally and no longer apply. Subsequently, developers may use this information to determine any PCE breakages due to the

base-code changes in question. However, base-code changes that do not cause *any* advice application differences may also result in broken PCEs in the case that a PCE was *supposed* to match an added JPS. Our approach can detect broken PCEs in such situations. Also, our approach suggests broken PCEs based on the breakage likelihood, whereas PCDiff will display *all* PCEs containing JPS differences.

[14] automatically creates *analysis-based* PCEs [8] from traditional named-based ones, such as those in AspectJ, which may avoid fragility issues. However, there is no round trip support to convert these PCEs back in cases where they do need to change.

[51] is geared towards aspect *mining*, i.e., converting non-AO programs to AO ones, which also works on maintaining *existing* AO systems. However, their approach is focused on incorporating missed JPSs into PCEs, whereas our approach is for detecting broken PCEs, either by inclusion *or* exclusion of JPSs, as the developer is typing. Moreover, the results of our tool are incorporated into an existing system (Mylyn) for focusing developer attention on particular software elements.

[30] is a change impact analysis for AOP. They detail an "Advice Invocation Change" (AIC) that indicates which PCEs are *affected* by new JPSs, but such an effect may not be a PCE breakage. Conversely, a new JPS that does not produce an AIC could also result in a broken PCE.

The AspectJ Development Tools (AJDT) is an Eclipse plug-in that contains several general, as well as IDE-specific, features to support AO development in Eclipse. These features include a built-in AspectJ compiler and a cross-references view. The AJDT also provides graphical (advice) markers that appear on the side of code editors. These markers depict information pertaining to advice that applies to the corresponding base code. Base code developers can click and/or hover over these markers to discover advice that may modify the behavior of the base at that point.

Although the AJDT is extremely effective in facilitating AspectJ software development by displaying current join point and PCE matching information, it does not display which PCEs do *not* match a given join point, which PCEs are likely to match in future versions of a given join point, and which PCEs are likely broken due to a particular join point.

PointcutDoctor [28] is an Eclipse plug-in that augments the AJDT to generalize (or *relax*) PCEs using several developer-minded heuristics. It exposes join points that were unintentionally *not selected* by PCEs, however, it does not inform the developer of situations where join points are unintentionally *selected*. These are join points that are selected by PCEs because the PCE was originally too general. Moreover, the standard heuristics it employs are not derived from the specific system at hand but, rather, general knowledge, including standard coding conventions, which may be

subjective, as well as subject to change. It does not analyze how PCEs evolve over versions and the join points that are selected by them in each version.

6. Conclusion and Future Work

We have detailed an approach that detects likely broken PCEs due to base-code changes in evolving Aspect-Oriented software using structural commonality. The automated approach programmatically manipulates the Mylyn DOI model in Eclipse while a developer edits an AspectJ project. It brings to the base-code developer's attention possibly broken PCEs, while likely unbroken PCEs move to the background, all in a standardized, consistent, and variably invasive way as the developer is typing. A developer can then take action to either alter the PCE or conform the base-code to the conventions set forth by the PCE. We have also shown how a recommendation system can be integrated into Mylyn, paving a way for future researchers to consider such integration and software developers to reap the benefits of that research.

Our empirical evaluation demonstrated that our approach is effective in bringing broken PCEs to light while suppressing unbroken PCEs, with such PCEs having DOI values that are greater than the average DOI value of unbroken ones, indicating a significant difference in final DOI value between the two kinds of pointcuts (i.e., broken and unbroken). The corresponding tool is publicly available for download as an extension to the popular Mylyn Eclipse plug-in.

In the future, we plan to persist patterns along with the Mylyn context and reuse them as to avoid rebuilding them if there are no changes in the base-code between task activations. This may have a performance impact in certain situations. Furthermore, the analysis phase commences only when a Mylyn task is activated, which occurs when a developer starts to work on a particular bug or feature. We plan to further investigate the optimal time to reanalyze the base-code. In addition, we will explore analyzing non-advice-bound PCEs for situations where PCEs are provided as "hooks" into modules per [15]. Such PCEs are not currently analyzed by the AJDT.

It would also be interesting to apply the Mylyn DOI model to AJDT markers, which appear on the side of the editor. Currently, markers are either fully present (solid) or not present at all. A Mylyn controlled marker would have different shades depending on the likelihood of the PCEs being broken, which would be dictated by Fraglight. This may help base-code developers pay attention to problematic PCEs earlier. Moreover, we plan to investigate the best way to present base-code developer guidance in how to fix broken PCEs (positive and negative PCE change predictions).

As detailed in Section 3, FRAGLIGHT contains a number of user-defined configuration parameters. These include maximum analysis depth (Section 3.2.1) and high and low confidence thresholds (Section 3.2.4). It would be optimal to assist developers with properly setting values for these parameters. As such, in the future, we plan to more thoroughly assess trade-offs between analysis depth and prediction time. A heuristic may be used to automatically adjust the analysis depth depending on the project size, e.g., smaller projects can be analyzed more thoroughly and vice-versa. An inductive machine learning algorithm may be employed in a feedback system to find the best confidence threshold values on a per-project or per-Mylyn context basis.

When FRAGLIGHT programmatically manipulates the Mylyn DOI, the amount by which the interestingness level is changed is solely dictated by the Mylyn framework. In the future, we plan to investigate varying the PCE interestingness level amount during manipulation, perhaps making it proportional to the degree of change confidence. This may entail invoking the appropriate Mylyn API several times per prediction depending on the change confidence value.

We also plan to evaluate FRAGLIGHT's DOI manipulation effectiveness in combination with other events that may affect the DOI, e.g., file navigation. Depending on the findings, we may derive best practices for editing base-code while FRAGLIGHT is activated, or adjust our algorithm to compensate for other events affecting the DOI model. A user study may be employed to assess how developers are likely to interact with FRAGLIGHT's suggestions, as well as the effectiveness of Mylyn's visualization in an AOP development setting.

Acknowledgments

This work was supported in part by the National Science Foundation [OISE-1015773] and the Japan Society for the Promotion of Science [SP10024]. We would like to thank Phil Greenwood, Amir Aryani, Sai Zhang, and Yu Lin for their answers to our many technical- and research-related questions and for referring us to related work.

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, Aspect oriented programming, in: European Conference on Object-Oriented Programming, Vol. 1241 of Lecture Notes in Computer Science, Springer-Verlag, 1997, pp. 220–242.

[2] G. C. Murphy, R. J. Walker, E. L. A. Baniassad, M. P. Robillard, A. Lai, M. A. Kersten, Does aspect-oriented programming work?, Commun. ACM 44 (10) (2001) 75–77. doi:10.1145/383845.383862.

[3] M. Kersten, G. C. Murphy, Atlas: A case study in building a web-based learning environment using aspect-oriented programming, in: ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '99, ACM, New York, NY, USA, 1999, pp. 340–352. doi:10.1145/320384.320421.

[4] M. Lippert, C. V. Lopes, A study on exception detection and handling using aspect-oriented programming, in: International Conference on Software Engineering, ICSE '00, ACM, New York, NY, USA, 2000, pp. 418–427. doi:10.1145/337180.337229.

[5] R. Walker, E. Baniassad, G. Murphy, An initial assessment of aspect-oriented programming, in: International Conference on Software Engineering, 1999, pp. 120–130.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An Overview of AspectJ, in: European Conference on Object-Oriented Programming, Vol. 2072 of Lecture Notes in Computer Science Lecture Notes in Computer Science, Springer Berlin/Heidelberg, 2001, pp. 327–354. doi:10.1007/3-540-45337-7.

[7] C. Koppen, M. Störzer, PCDiff: Attacking the fragile pointcut problem, in: K. Gybels, S. Hanenberg, S. Herrmann, J. Wloka (Eds.), European Interactive Workshop on Aspects in Software, 2004, pp. 1–8.

[8] T. Aotani, H. Masuhara, SCope: an aspectj compiler for supporting user-defined analysis-based pointcuts, in: International Conference on Aspect-Oriented Software Development, ACM, 2007, pp. 161–172.

[9] K. Ostermann, M. Mezini, C. Bockisch, Expressive Pointcuts for Increased Modularity, in: European Conference on Object-Oriented Programming, Vol. 3586 of Lecture Notes in Computer Science, Springer Berlin/Heidelberg, 2005, pp. 214–240. doi:10.1007/11531142_10.

[10] W. Cazzola, S. Pini, M. Ancona, Design-Based Pointcuts Robustness Against Software Evolution, in: W. Cazzola, S. Chiba, Y. Coady, G. Saake (Eds.),

Workshop on Reflection, AOP, and Meta-Data for Software Evolution, Fakultät für Informatik, Universität Magdeburg, 2006, pp. 35–45.

[11] K. Sakurai, H. Masuhara, Test-based pointcuts for robust and fine-grained join point specification, in: International Conference on Aspect-Oriented Software Development, ACM, Washington, DC, USA, 2008, pp. 96–107. doi:10.1145/1353482.1353494.

[12] K. Klose, K. Ostermann, Back to the Future: Pointcuts as Predicates over Traces, in: C. Clifton, R. Lämmel, G. T. Leavens (Eds.), International Workshop on Foundations of Aspect-Oriented Languages, 2005, p. 33.

[13] L. M. Seiter, Role annotations and adaptive aspect frameworks, in: International Workshop on Linking aspect technology and evolution, ACM, New York, NY, USA, 2007, p. 3.

[14] L. Wang, T. Aotani, M. Suzuki, Improving the Quality of AspectJ Application: Translating Name-Based Pointcuts to Analysis-Based Pointcuts, in: International Conference on Quality Software, 2014, pp. 27–36. doi:10.1109/QSIC.2014.34.

[15] J. Aldrich, Open Modules: Modular Reasoning About Advice, in: European Conference on Object-Oriented Programming, Vol. 3586 of Lecture Notes in Computer Science, Springer Berlin/Heidelberg, 2005, pp. 144–168.

[16] S. Gudmundson, G. Kiczales, Addressing Practical Software Development Issues in AspectJ with a Pointcut Interface, in: Workshop on Advanced Separation of Concerns at the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2001, pp. 1–6.

[17] R. Khatchadourian, J. Dovland, N. Soundarajan, Enforcing behavioral constraints in evolving aspect-oriented programs, in: International Workshop on Foundations of Aspect-Oriented Languages, ACM, New York, NY, USA, 2008, pp. 19–28.

[18] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, H. Rajan, Modular Software Design with Crosscutting Interfaces, IEEE Softw. 23 (2006) 51–60. doi:10.1109/MS.2006.24.

[19] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, H. Rajan, Information hiding interfaces for aspect-oriented design, in: ACM SIGSOFT

Symposium on the Foundations of Software Engineering, ACM, New York, NY, USA, 2005, pp. 166–175. doi:10.1145/1081706.1081734.

[20] M. Bagherzadeh, H. Rajan, G. T. Leavens, S. Mooney, Translucid contracts: expressive specification and modular verification for aspect-oriented interfaces, in: International Conference on Aspect-Oriented Software Development, ACM, New York, NY, USA, 2011, pp. 141–152. doi:10.1145/1960275.1960293.

[21] K. Hoffman, P. Eugster, Bridging Java and AspectJ through explicit join points, in: International Symposium on Principles and Practice of Programming in Java, ACM, New York, NY, USA, 2007, pp. 63–72. doi:10.1145/1294325.1294335.

[22] H. Rajan, G. Leavens, Ptolemy: A Language with Quantified, Typed Events, in: European Conference on Object-Oriented Programming, Vol. 5142 of Lecture Notes in Computer Science, Springer Berlin/Heidelberg, 2008, pp. 155–179.

[23] E. Bodden, É. Tanter, M. Inostroza, Join point interfaces for safe and flexible decoupling of aspects, ACM Transactions on Software Engineering and Methodology 23 (1) (2014) 7.

[24] F. Steimann, T. Pawlitzki, S. Apel, C. Kästner, Types and modularity for implicit invocation with implicit announcement, ACM Transactions on Software Engineering and Methodology 20 (1) (2010) 43.

[25] A. Kellens, K. Mens, J. Brichau, K. Gybels, Managing the evolution of aspect-oriented software with model-based pointcuts, in: European Conference on Object-Oriented Programming, ECOOP'06, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 501–525. doi:10.1007/11785477_28.

[26] Oracle Corporation, Java Programming Language: Enhancements in JDK 5: Annotations, last checked: January 23, 2015. (2010).
URL http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html

[27] A. Clement, A. Colyer, M. Kersten, Aspect-oriented programming with ajdt, in: ECOOP Workshop on Analysis of Aspect-Oriented Software, 2003, pp. 1–6.

[28] L. Ye, K. D. Volder, Tool support for understanding and diagnosing pointcut expressions, in: International Conference on Aspect-Oriented Software Development, AOSD '08, ACM, New York, NY, USA, 2008, pp. 144–155. doi:10.1145/1353482.1353500.

[29] J. Wloka, R. Hirschfeld, J. Hänsel, Tool-supported refactoring of aspect-oriented programs, in: International Conference on Aspect-Oriented Software Development, AOSD '08, ACM, New York, NY, USA, 2008, pp. 132–143. doi:10.1145/1353482.1353499.

[30] J. Zhao, Change impact analysis for aspect-oriented software evolution, in: International Workshop on Principles of Software Evolution, ACM, New York, NY, USA, 2002, pp. 108–112. doi:10.1145/512035.512060.

[31] R. Khatchadourian, P. Greenwood, A. Rashid, G. Xu, Pointcut rejuvenation: Recovering pointcut expressions in evolving aspect-oriented software, Transactions on Software Engineering 38 (3) (2012) 642–657.

[32] M. Kersten, G. C. Murphy, Mylar: a degree-of-interest model for IDEs, in: International Conference on Aspect-Oriented Software Development, ACM, New York, NY, USA, 2005, pp. 159–168. doi:10.1145/1052898.1052912.

[33] The Eclipse Foundation, The mylyn task and application lifecycle management framework, last checked January 20, 2015. (2015).
URL http://www.eclipse.org/mylyn

[34] M. Kersten, G. C. Murphy, Using task context to improve programmer productivity, in: ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM, New York, NY, USA, 2006, pp. 1–11. doi:10.1145/1181775.1181777.

[35] S. Clarke, R. J. Walker, Composition patterns: An approach to designing reusable aspects, in: International Conference on Software Engineering, ICSE '01, IEEE Computer Society, Washington, DC, USA, 2001, pp. 5–14.
URL http://dl.acm.org/citation.cfm?id=381473.381474

[36] R. Khatchadourian, A. Rashid, H. Masuhara, T. Watanabe, Detecting broken pointcuts using structural commonality and degree of interest, in: International Conference on Automated Software Engineering, ASE '15, IEEE Computer Society, Washington, DC, USA, 2015, pp. 641–646. doi:10.1109/ASE.2015.80.

[37] R. Khatchadourian, A. Rashid, H. Masuhara, T. Watanabe, Fraglight: Shedding light on broken pointcuts in evolving aspect-oriented software, in: Companion publication of the ACM SIGPLAN conference on Systems, Programming, and Applications: Software for Humanity, SPLASH Companion 2015, ACM, New York, NY, USA, 2015, pp. 17–18. doi:10.1145/2814189.2814195.

[38] R. Delamare, B. Baudry, S. Ghosh, Y. L. Traon, A test-driven approach to developing pointcut descriptors in aspectj, in: International Conference on Software Testing Verification and Validation, ICST '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 376–385. doi:10.1109/ICST.2009.41.

[39] H. Masuhara, G. Kiczales, C. Dutchyn, A Compilation and Optimization Model for Aspect-Oriented Programs, in: International Conference on Compiler Construction, Vol. 2622 of Lecture Notes in Computer Science, Springer Berlin/Heidelberg, 2003, pp. 46–60. doi:10.1007/3-540-36579-6_4.

[40] S. Apel, How AspectJ is Used: An Analysis of Eleven AspectJ Programs, Journal of Object Technology 9 (1) (2010) 117–142.

[41] B. Dagenais, S. Breu, F. W. Warr, M. P. Robillard, Inferring structural patterns for concern traceability in evolving software, in: International Conference on Automated Software Engineering, IEEE/ACM, 2007, pp. 254–263.

[42] IBM, Eclipse Documentation - Interface IElementChangeListener, last checked January 23, 2015. (2012).
URL http://help.eclipse.org/juno/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ui/texteditor/IElementStateListener.html

[43] IBM, Eclipse Documentation - Interface IJavaElementDelta, last checked January 23, 2015. (2012).
URL http://help.eclipse.org/galileo/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/IJavaElementDelta.html

[44] B. Fluri, M. Wursch, M. Pinzger, H. Gall, Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction, Transactions on Software Engineering 33 (11) (2007) 725–743. doi:10.1109/TSE.2007.70731.

[45] J. Dean, D. Grove, C. Chambers, Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis, in: European Conference on Object-Oriented Programming, Vol. 952 of Lecture Notes in Computer Science, Springer Berlin/Heidelberg, 1995, pp. 77–101.

[46] C. L. Forgy, Rete: a fast algorithm for the many pattern/many object pattern match problem, Artificial Intelligence 19 (1982) 324–341.

[47] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. C. Filho, F. Dantas, Evolving software product lines with aspects: an empirical study on design stability, in: International Conference on Software Engineering, ACM, New York, NY, USA, 2008, pp. 261–270. doi:10.1145/1368088.1368124.

[48] P. Greenwood, T. T. Bartolomei, E. Figueiredo, M. Dósea, A. F. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, A. Rashid, On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study, in: European Conference on Object-Oriented Programming, Vol. 4609 of Lecture Notes in Computer Science, Springer Berlin/Heidelberg, 2007, pp. 176–200.

[49] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: elements of reusable object-oriented software, Addison-Wesley Professional, 1995.

[50] R. P. Buse, C. Sadowski, W. Weimer, Benefits and barriers of user evaluation in software engineering research, in: ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '11, ACM, New York, NY, USA, 2011, pp. 643–656. doi:10.1145/2048066.2048117.

[51] T. T. Nguyen, H. V. Nguyen, H. A. Nguyen, T. N. Nguyen, Aspect recommendation for evolving software, in: International Conference on Software Engineering, ACM, 2011, pp. 361–370.