Fraglight: Shedding Light on Broken Pointcuts in Evolving Aspect-Oriented Software

Raffi Khatchadourian
City University of New York, USA
rkhatchadourian@citytech.cuny.edu

Awais Rashid
Lancaster University, UK
awais@comp.lancs.ac.uk

Hidehiko Masuhara
Tokyo Institute of Technology, Japan
masuhara@acm.org

Takuya Watanabe
Edirium K.K., Japan
sodium@edirium.co.jp

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 demonstrate an automated tool called FRAGLIGHT that recommends a set of pointcuts that are likely to require modification due to a particular base-code change. The underlying approach is rooted in harnessing unique and arbitrarily deep structural commonality between program elements corresponding to join points selected by a pointcut in a particular software version. Patterns describing such commonality are used to recommend pointcuts that have potentially broken with a degree of confidence as the developer is typing. Our tool is implemented as an extension to the Mylyn Eclipse IDE plug-in, which maintains focused contexts of entities relevant to a task.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement

Keywords Aspect-Oriented programming, software evolution

1. Introduction

Although using Aspect-Oriented Programming (AOP) [9] can be beneficial to developers in many ways, such systems have 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 [10], 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, i.e., they may fail to select or inadvertently select new places in the program's execution, a problem known as pointcut fragility [11]. Deciding which PCEs have broken is a daunting venture, especially in large and complex systems. In software with many PCEs, seemingly innocuous 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).

Several approaches offer tool-support for detecting broken PCEs. The AspectJ Development Tools (AJDT) [1], 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. Ye and Volder [13] augment 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. Wloka et al. [12] automatically fix PCEs broken by refactorings, however, manual base-code edits may also break PCEs. In our previous work [7], we periodically suggest join points that may require inclusion by a PCE. Yet, developers must *manually* detect broken PCEs, as well as determine how frequently to check.

We demonstrate an automated approach that recommends PCEs that are likely to require modification due to base-code changes [8]. Our approach has been implemented as an automated AspectJ source-level inferencing tool called FRAGLIGHT, which is a plug-in to the popular Eclipse IDE (`http://eclipse.org`). 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 of the Mylyn context [5].

Mylyn (`http://eclipse.org/mylyn`) is a standard Eclipse plug-in that facilitates software evolution by focusing graphical components of the IDE so that only artifacts related the currently active task are revealed to the developer [6]. The context is comprised of the relevant elements, 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 the file, edits

SPLASH Companion'15, October 25–30, 2015, Pittsburgh, PA, USA

http://dx.doi.org/10.1145/2814189.2814195

the file) when working on a task, the more interesting the element is deemed to be, and vice-versa.

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 makes PCEs that are *more* likely to be broken *more interesting*, i.e., by *increasing* its DOI value, while implicitly making PCEs that are *less* likely to be 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 the attention of the developer are presented more prominently in the IDE than ones that are less likely. The developer can then make alteration decisions based on FRAGLIGHT's recommendations, possibly adjusting the PCE or the base-code to rectify the problem. 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, making these systems easier to maintain.

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. In [7], we showed that the majority of program elements corresponding to join points selected by a PCE in one base-code version shared such characteristics between them, and that these relationships persisted in subsequent versions. FRAGLIGHT uses this premise to detect broken PCEs on-the-fly.

2. Implementation

FRAGLIGHT is implemented as a relation provider extension to the standard Mylyn Eclipse plug-in. The JayFX fact extractor [2], which we extended for use with modern Java languages and AspectJ, is used to generate "facts," using class hierarchical analysis (CHA) [3], 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.

The AJDT is used to conservatively associate a PCE with structural properties. For a given PCE, the AJDT produces the Java program elements, e.g., method declarations, method calls, field sets, correlated with selected join points. Pattern extraction and pattern matching are implemented via the Drools Rules Engine (`http://drools.org`), which uses a modified RETE algorithm [4]. 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`).

Due to a present Eclipse framework limitation, the AJDT is not able to reconcile AspectJ code without first saving it to disk (see `http://bugs.eclipse.org/bugs/show_bug.cgi?id=310046`). As such, our prototype implementation saves the current editor buffer every time it detects a new join point being added. In the future, we plan to add in-memory reconciliation to the AJDT so that the use of FRAGLIGHT would be more developer-friendly.

FRAGLIGHT's implementation is unique in that, as far as we know, it is the first change prediction tool to be integrated with Mylyn. Mylyn is a natural vehicle for change predictions as it programmatically determines the next elements of the developer's interest through IDE interactions. FRAGLIGHT's statistical approach adds to Mylyn by using static analysis to manipulate the DOI.

In a preliminary empirical evaluation, we ran our tool on projects ranging from $\approx 1.4-6$K non-blank, non-commented lines of code (LOC). We found the analysis time to be practical, averaging ≈ 1.05 secs per KLOC and ≈ 0.14 secs per PCE. Moreover, we found the prediction time to be, on average, ≈ 2.61 per added join point, which is also practical, especially since the tool runs in the background. Lastly, we found that the average DOI value of PCEs that actually broke were, on average, 2.5 times greater than those that did not break, which is promising. In the future, we plan to administer a full empirical evaluation.

3. Demonstration

We will portray a hypothetical developer session using FRAGLIGHT. The example software will be of modest yet understandable size. It will include two PCEs, each representing different crosscutting concerns. We will then evolve the software in a series of ways, including bug fixes and feature additions. During these changes, one of the two PCEs will break. Will will see how FRAGLIGHT manipulates the DOI so that the broken PCE programmatically becomes more prominent in the IDE, while the unbroken PCE becomes less prominent. We will also bring to light the internal pattern matching being performed by FRAGLIGHT in the background.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. OISE-1015773 and the Japan Society for the Promotion of Science under Grant No. 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. We would also like to thank the anonymous reviewers for their extremely useful comments and suggestions.

References

[1] A. Clement, A. Colyer, and M. Kersten. Aspect-oriented programming with ajdt. In *ECOOP Workshop on Analysis of Aspect-Oriented Software*, 2003.

[2] B. Dagenais, S. Breu, F. W. Warr, and M. P. Robillard. Inferring structural patterns for concern traceability in evolving software. In *International Conference on Automated Software Engineering*, 2007.

[3] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *European Conference on Object-Oriented Programming*, 1995.

[4] C. L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 1982.

[5] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for IDEs. In *International Conference on Aspect-Oriented Software Development*, 2005.

[6] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2006.

[7] R. Khatchadourian, P. Greenwood, A. Rashid, and G. Xu. Pointcut rejuvenation: Recovering pointcut expressions in evolving aspect-oriented software. *IEEE Transactions on Software Engineering*, 2012.

[8] R. Khatchadourian, A. Rashid, H. Masuhara, and T. Watanabe. Detecting broken pointcuts using structural commonality and degree of interest. In *International Conference on Automated Software Engineering*, 2015.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect oriented programming. In *European Conference on Object-Oriented Programming*, 1997.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *European Conference on Object-Oriented Programming*, 2001.

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

[12] J. Wloka, R. Hirschfeld, and J. Hänsel. Tool-supported refactoring of aspect-oriented programs. In *International Conference on Aspect-Oriented Software Development*, 2008.

[13] L. Ye and K. D. Volder. Tool Support For Understanding and Diagnosing Pointcut Expressions. In *International Conference on Aspect-Oriented Software Development*, 2008.