Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

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

A R T I C L E I N F O

Article history:
Received 7 September 2016
Received in revised form 6 June 2017
Accepted 26 June 2017
Available online xxxx

Keywords:
Software development environments
Software maintenance
Software tools

A B S T R A C T

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 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).

1. Introduction

Although using Aspect-Oriented Programming (AOP) [1] can be beneficial to developers in many ways [2–5], such systems have the potential for new problems unique to the paradigm. A key construct that allows code to be situated in a

* Corresponding author.
E-mail addresses: raffi.khatchadourian@hunter.cuny.edu (R. Khatchadourian), awais@comp.lancs.ac.uk (A. Rashid), masuhara@acm.org (H. Masuhara), sodium@edirium.co.jp (T. 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.

http://dx.doi.org/10.1016/j.scico.2017.06.011
0167-6423/© 2017 Elsevier B.V. All rights reserved.

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 [8–14], limiting where advice may apply [15,16], or enforcing constraints on advice application [17–20]. Others make advice applicability more explicit [21] or do not use PCEs [22–24]. 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