The Journal of Systems and Software 82 (2009) 668–684

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Using aspect orientation in legacy environments for reverse engineering using dynamic analysis—An industrial experience report ☆

Bram Adams [a,*], Kris De Schutter [b], Andy Zaidman [c], Serge Demeyer [d], Herman Tromp [a], Wolfgang De Meuter [b]

[a] Ghent University, INTEC, Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium
[b] Vrije Universiteit Brussel, PROG, Pleinlaan 2, B-1050 Brussels, Belgium
[c] Delft University of Technology, SERG, Mekelweg 4, 2628CD Delft, The Netherlands
[d] University of Antwerp, LORE, Middelheimlaan 1, B-2020 Antwerp, Belgium

ARTICLE INFO

Article history:
Received 11 February 2007
Received in revised form 25 September 2008
Accepted 25 September 2008
Available online 8 October 2008

Keywords:
Dynamic analysis
Aspect-oriented programming
Industrial case study
Program comprehension C

ABSTRACT

This paper reports on the challenges of using aspect-oriented programming (AOP) to aid in re-engineering a legacy C application. More specifically, we describe how AOP helps in the important reverse engineering step which typically precedes a re-engineering effort. We first present a comparison of the available AOP tools for legacy C code bases, and then argue on our choice of Aspicere, our own AOP implementation for C. Then, we report on Aspicere's application in reverse engineering a legacy industrial software system and we show how we apply a dynamic analysis to regain insight into the system. AOP is used for instrumenting the system and for gathering the data. This approach works and is conceptually very clean, but comes with a major quid pro quo: integration of AOP tools with the build system proves an important issue. This leads to the question of how to reconcile the notion of modular reasoning within traditional build systems with a programming paradigm which breaks this notion.

1. Introduction

Legacy software is omnipresent: software that is still very useful to an organisation – quite often even *indispensable* – but the evolution of which becomes too great a burden Bennett (1995). This burden can be caused by an increase in the complexity brought on by the normal evolution of the system (Sneed, 2005; Brodie and Stonebraker, 1995; Moise and Wong, 2003; Carver and Montes de Oca, 1998; Demeyer et al., 2003; Lehman and Belady, 1985). Classic symptoms include

- a lack of experienced developers or maintainers,
- a lack of up-to-date documentation, and
- technology that does not reflect the current (business) environment.

To counter this phenomenon, a number of solutions to cope with evolution have been proposed (Bennett, 1995; Sneed, 1996) in the field of re-engineering (Chikofsky and Cross, 1990). When applying these countermeasures in a reliable, economically sound and swift fashion, the software engineer would ideally like to have (1) a *deep insight* into the application in order to start his/her re-engineering operation (Sneed, 2004; Carver and Montes de Oca, 1998; Lehman, 1998) and (2) a well-covering (set of) regression test(s) to check whether the adaptations made are behavior-preserving (Demeyer et al., 2003; Ducasse et al., 2006). In practice, legacy applications seldom have up to date documentation (Moise and Wong, 2003), nor do they have such a set of tests.

For all these reasons we are interested in the re-engineering of legacy E-type systems (*software systems that solve a problem or implement a computer application in the real world* (Lehman, 1996)). Recent research (Mens and Tourwé, 2008; Colyer and Clement, 2004; Lämmel and De Schutter, 2005) suggests that aspect-oriented programming (AOP) (Kiczales et al., 1997) plays an important role in this effort as it provides a modularised way to change the existing behaviour of a system without having to destructively modify that system's source code in any way. The modularity provides us with opportunities for re-engineering, while the non-invasiveness takes care of some of the psychological concerns associated with modifying business-critical source code.

☆ This article is an extension to our earlier paper *Regaining Lost Knowledge through Dynamic Analysis and Aspect Orientation*, published in the proceedings of the Conference on Software Maintenance and Re-engineering (CSMR'06) (Zaidman et al., 2006).

* Corresponding author. Tel.: +32 92643318; fax: +32 92643593.
E-mail addresses: Bram.Adams@ieee.org (B. Adams), kdeschut@vub.ac.be (K. De Schutter), a.e.zaidman@tudelft.nl (A. Zaidman), Serge.Demeyer@ua.ac.be (S. Demeyer), Herman.Tromp@ugent.be (H. Tromp), wdmeuter@vub.ac.be (W. De Meuter).

0164-1212/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2008.09.031

We have been looking at applying AOP in forward engineering (Lämmel and De Schutter, 2005; Schutter and Adams, 2007; De Schutter, 2006; Adams and Schutter, 2007), as have others (Bruntink et al., 2007; Bruntink et al., 2005; Mens and Tourwé, 2008), with success. Different from these, this paper takes a first look at an opportunity for AOP in a reverse engineering setting. Reverse engineering is the essential first step in the re-engineering process, and has been reported to take up to 60% of the required effort (Corbi, 1990).

As part of our research in the ARRIBA[1] project, our focus is on industrial legacy systems. Considering this, we choose to use dynamic analysis for our reverse engineering process. This choice is instigated by the fact that dynamic analysis allows us to follow a goal-oriented strategy, i.e., it lets us analyze only those parts of the system that we are really interested in (Zaidman, 2006). This goal-oriented strategy is certainly warranted considering the scale of typical legacy applications. Furthermore, it puts us in the position to report on the benefits of using dynamic analysis in a large-scale industrial legacy setting, of which reports are scarce (e.g., Eisenbarth et al., 2003; Callo Arias et al., in press). In order to enable this dynamic analysis, we introduce a simple tracing aspect into an industrial system. Given that we only need to collect a representative trace of the running application in order for the dynamic analysis to work, we could also have opted for dedicated tools such as DTRACE (Cantrill et al., 2004) or ATOM (Srivastava and Eustace, 1994). There are two reasons we do not do this. One is that we are looking at AOP as a tool in the *entire* re-engineering chain and *not* limited to a particular reverse engineering technique. As proposed by De Roover et al. (2006), aspects can generate *reverse*-engineering results in such a way that *re*-engineering aspects can exploit these results to steer their re-engineering tasks. In this respect AOP is more interesting as it is more generally applicable than the aforementioned tools. The second reason is that we also need to consider how to get our aspects applied in real-life systems. As this paper shows, even for something as simple as a tracing aspect, *this is not trivial*. Indeed, as the prototypical example of an extremely scattered aspect, a tracing aspect actually provides us with something of a stress test with respect to the support of aspects in the legacy system.

The experiment reported on in this paper is therefore on a mid-size real-life system which has accumulated a mix of Kernighan & Ritchie (K&R) Kernighan and Ritchie (1978) as well as ANSI-C style code. This has an impact on our choice of AOP tool, which this paper will also take into careful consideration.

In short, the contributions of this paper are

- a comprehensive overview of AOP tools for the C programming language,
- the introduction of a new AOP tool which fits our re-engineering goals,
- the application of a dynamic analysis on an industrial legacy application,
- a discussion of some of the problems found when applying AOP in a legacy setting.

The structure of the remainder of this paper is as follows: Section 2 explores the possible AOP tools for legacy C systems. As we will see there is none that fits the bill and so Section 3 introduces a tool of our own which has been created according to our re-engineering goals. Next, Section 4 describes an actual application of AOP in an industrial environment by showcasing a dynamic analysis approach; we present the actual experiment, including the aspect we apply, the results we get from the analysis, and the validation of those results with the system's developers. Section 5 then discusses the problems encountered while trying to apply AOP to this system. Section 6 describes the threats to validity. Section 7 describes the related work, followed by Section 8 which rounds up the discussion with our conclusions.

2. AOP tools for legacy C applications

As discussed by Mens and Tourwé (2008), aspect extraction and evolution are two crucial activities when re-engineering a system using aspects. Failure or success of AOP for re-engineering depends to a large extent on sufficient aspect language support. Without this, the re-engineered system risks becoming unmaintainable and even less manageable than the original system.

This section first provides a brief introduction on AOP, before narrowing the focus to requirements for aspect languages for legacy systems. We then discuss the aspect languages for C which existed at the time of starting our research. Finally, we compare the aspect languages.

2.1. Aspect-oriented programming

Aspect-oriented programming (AOP) modularises so-called "crosscutting concerns" (CCCs) (Kiczales et al., 1997). When developers implement these concerns using traditional programming language techniques, two undesired phenomena typically crop up in the source code: scattering and tangling. The former corresponds to implementation fragments of a concern (e.g., caching), which occur at many places throughout the source code. Hence, a change to the implementation of the concern requires changes at many locations in the source code, which is tedious, error-prone and hampers understandability. The situation is even worse, because at each location where a concern fragment occurs, it may be tangled (mixed) with fragments of other concerns. This means that programmers need to understand the interplay between multiple concerns before being able to modify the caching concern. AOP deals with these undesirable program properties by extracting crosscutting concerns in a new kind of modules: aspects.

To date, AspectJ is still the primary aspect language in existence, both in research and in practice. This is an aspect language for Java which has introduced the concepts of advice, pointcut, join points, etc. An aspect is similar to a class or module, but can contain "advice", which consists of a "pointcut"[2] and an "advice body". According to the most common school, the implementation of crosscutting concerns is extracted from the "base code". The latter corresponds to the implementation of the main concerns, the so-called "dominant decomposition" which forms the backbone of the whole system. CCC implementation fragments are separated from the base code and localised into (possibly) multiple advice bodies of an aspect.

Code separation is only one part of the effort required to resolve scattering and tangling. One still needs to specify at which moments during the base program execution an advice body should be invoked. Instead of embedding explicit calls to advice within the base code, an advice is invoked automatically once a condition (pointcut) is satisfied. This inversion of dependencies (Martin and Nordberg, 2001) forms the core idea behind AOP. The moments in time when advice can be triggered are called "join points", as this is where the main concern(s) and a CCC join each other. Established kinds of join points are method calls and executions, variable access and manipulation, etc. A pointcut can make use of program structure, name patterns, dynamic program state, etc. to describe the intended set of join points. It is, for example, possible

[1] *Architectural Resources for the Restructuring and Integration of Business Applications*. More info on this project at http://arriba.vub.ac.be/.

[2] Sometimes abbreviated to "PCD", for "pointcut designator".