The Journal of Systems and Software 83 (2010) 391–404

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Identification of refactoring opportunities introducing polymorphism

Nikolaos Tsantalis, Alexander Chatzigeorgiou *

Department of Applied Informatics, University of Macedonia, 54006 Thessaloniki, Greece

ARTICLE INFO

Article history:
Received 14 November 2008
Received in revised form 3 September 2009
Accepted 3 September 2009
Available online 11 September 2009

Keywords:
Refactoring
Polymorphism
State/Strategy design pattern
Object-oriented design

ABSTRACT

Polymorphism is one of the most important features offered by object-oriented programming languages, since it allows to extend/modify the behavior of a class without altering its source code, in accordance to the *Open/Closed Principle*. However, there is a lack of methods and tools for the identification of places in the code of an existing system that could benefit from the employment of polymorphism. In this paper we propose a technique that extracts refactoring suggestions introducing polymorphism. The approach ensures the behavior preservation of the code and the applicability of the refactoring suggestions based on the examination of a set of preconditions.

1. Introduction

Polymorphism has been widely recognized as one of the most important features of object-oriented programming languages. As polymorphism we refer to subtype polymorphism which according to Day et al. (1995) allows code written in terms of some type *T* to actually work for all subtypes of *T*. The main advantage of polymorphism is that it allows client classes to depend on abstractions (Gamma et al., 1995; Martin, 2003). An abstraction (abstract class or interface) can be extended by adding new subclasses that conform to its interface (i.e. override its abstract methods). However, the client classes that depend on abstractions do not have to change in order to take advantage of the behavior defined in the new subclasses.

Despite the sedulous teaching of polymorphism in object-oriented programming courses and its detailed presentation and discussion in books appealing to professionals, *state-checking* is often employed as an alternative approach to polymorphism in order to simulate *late binding* and *dynamic dispatch*. State-checking manifests itself as conditional statements that select an execution path either by comparing the value of a variable representing the current state of an object with a set of named constants, or by retrieving the actual subclass type of a reference through *RunTime Type Identification* (RTTI) mechanisms. The aforementioned symptoms usually result from either poor quality of the initial design or software aging (Parnas, 1994) caused by requirement changes that were not anticipated in the original design. State-checking introduces additional complexity due to conditional statements consisting of many cases and code duplication due to conditional statements scattered in many different places of the system that perform state-checking on the same cases for different purposes (Fowler et al., 1999). As a result, the maintenance of multiple state-checking code fragments operating on common states may require significant effort and introduce errors.

Although the employment of polymorphism in object-oriented systems is considered as an important design quality indicator, there is a lack of tools that either identify state-checking cases in an existing system or eliminate them by applying the appropriate refactorings on source code. To this end, we propose a technique for the identification and elimination of state-checking problems in Java projects that has been implemented as an Eclipse plug-in. An advantage of the proposed approach over metric-based approaches is the fact that all identified problems are actual cases of state-checking rather than ordinary conditional statements. Moreover, the examination of a set of preconditions ensures that the refactoring suggestions are both applicable and behavior-preserving.

The approach can be considered as semi-automatic, since after the extraction of the refactoring suggestions the designer is responsible for deciding whether a state-checking case should be eliminated or not based on conceptual and design quality criteria. Regarding the automation of the identification process, the main difference of the proposed technique with state-of-the-art Integrated Development Environments (IDEs) offering refactoring support (e.g. Eclipse 3.5, Netbeans 6.7, IntelliJ IDEA 8.1, Visual Studio 2008 along with Refactor! Pro 2.5) is that IDEs determine which refactorings are applicable based on the selection of a code fragment by the developer, while the proposed technique identifies

* Corresponding author. Tel.: +30 2310 891886; fax: +30 2310 891791.
E-mail addresses: nikos@java.uom.gr (N. Tsantalis), achat@uom.gr (A. Chatzigeorgiou).

0164-1212/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2009.09.017

refactoring opportunities without requiring any human intervention. Moreover, the proposed technique assists the designer to determine the effectiveness of the identified refactoring opportunities by grouping them according to their relevance and sorting them according to various quantitative characteristics.

The evaluation of the proposed technique consists of three parts. The first part presents the precision and recall of the approach by comparing the refactoring opportunities identified by an independent expert to the results of the proposed technique on various open-source projects. The second part of the evaluation investigates the impact of three quantitative factors on the decision of the independent expert to accept or reject the refactoring opportunities identified by the proposed technique. The last part refers to the scalability of the technique based on the computation time required for the extraction of refactoring suggestions on various open-source projects which differ in size characteristics.

The rest of the paper is organized as follows: Section 2 provides an overview of the related work. The proposed technique is thoroughly analyzed in Section 3, and Section 4 presents the tool that implements it. The results of the evaluation are discussed in Section 5. Finally, we conclude in section 6.

2. Related work

According to Gamma et al. (1995), polymorphism simplifies the definitions of clients, decouples objects from each other, and lets them vary their relationships to each other at runtime. To this end, polymorphism plays a key role to the structure and behavior of most design patterns. In the literature of object-oriented software engineering, several empirical studies have investigated the impact of polymorphism and design patterns on external quality indicators related with software maintenance.

Brito e Abreu and Melo (1996) have shown that Polymorphism Factor (Brito e Abreu, 1995), which is defined as the number of methods that override inherited methods divided by the maximum number of possible distinct polymorphic situations, has a moderate to high negative correlation with defect and failure densities as well as with rework. In other words, the appropriate use of polymorphism in an object-oriented design should decrease the defect density and rework. However, they have also supported that very high values of Polymorphism Factor (above 10%) are expected to reduce these benefits, since the understanding and debugging of a highly polymorphical hierarchy is much harder than the procedural counterpart.

Prechelt et al. (2001) conducted a controlled experiment to compare design pattern solutions to simpler alternatives in terms of maintenance. The subjects of the experiment were professional software engineers that were asked to perform a variety of maintenance tasks. The independent variables were the programs and change tasks, the program version (there were two different functional equivalent versions of each program, a pattern-based version and an alternative version with simpler solutions) and the amount of pattern knowledge of the participants. The dependent variables were the time taken for each maintenance task and the correctness (i.e. whether the solutions fulfilled the requirements of the task). In most of the cases the experimental results had shown positive effects from the use of design patterns, since maintenance time was reduced compared to the simpler alternative versions.

Ng et al. (2006) performed a controlled experiment on maintaining JHotDraw to study whether the introduction of additional patterns through program refactoring is beneficial regardless of the work experience of the maintainers. For this reason, they used two sets of subjects in their experiment, namely experienced and inexperienced maintainers. They compared two maintenance approaches where in the first approach the subjects performed the maintenance tasks directly on the original program, while in the second approach the subjects performed the maintenance tasks on a refactored version of the original program using additional design patterns to facilitate the required changes. The empirical results have shown that, to complete a maintenance task of perfective nature, the time spent even by the inexperienced maintainers on the refactored version was much shorter than that of the experienced subjects on the original version.

Ng et al. (2007) studied whether maintainers utilize deployed design patterns, and when they do, which tasks they more commonly perform. For this reason, they refined an anticipated change facilitated by the deployment of design patterns into three finer-grained maintenance tasks, namely adding new concrete participants, modifying the existing interfaces of a participant, and introducing a new client. They concluded that regardless of the type of tasks performed by maintainers when utilizing deployed design patterns for anticipated changes, the delivered code is significantly less faulty than the code developed without utilizing patterns.

Other empirical studies have shown that maintenance effort does not only depend on the design quality of a given program (as expressed by the employment of design principles or the existence of design patterns), but also on human factors such as the experience, skills and education of the software developers and maintainers.

Arisholm and Sjøberg (2004) performed a controlled experiment in order to investigate the effect of delegated versus centralized control style on the maintainability of object-oriented software. To this end, two categories of developers (namely experienced and inexperienced) performed several change tasks on two alternative designs that had a centralized and delegated control style, respectively. The results of the experiment have shown that the most experienced developers required less time to maintain the software with delegated control style than with centralized control style, while novice developers had serious problems in understanding the delegated control style and performed much better with the centralized control style. Consequently, they concluded that maintainability of object-oriented software depends, to a large extent, on the skill of the maintainers.

Du Bois (2006) performed a series of controlled experiments to investigate whether the application of two reengineering patterns (Demeyer et al., 2003), namely *Refactor to Understand* and *Split Up God Class*, can improve program comprehension. The experiment involving the decomposition of god classes verified that the particular education of the subject performing the comprehension task affects the way in which a god class is decomposed.

Wendorff (2001) reported on a large commercial project where the uncontrolled use of patterns has contributed to severe maintenance problems. The reasons causing the maintenance problems were that some pattern instances were misused by software developers who had not understood the rationale behind their employment, many software developers overestimated the future volatility of requirements and opted for patterns to build flexibility at the cost of an undesirable increase of complexity, the change of requirements over the lifetime of the project led some pattern instances to become obsolete, and finally some pattern instances were embellished with additional features which were not actually needed. Consequently, the inappropriate application of patterns may have a negative effect on flexibility and maintainability of object-oriented software.

Concerning performance, it is widely believed that the replacement of conditional logic by a polymorphic method call deteriorates performance due to the introduction of an additional indirection through the *virtual function table*. Demeyer (2005) investigated the performance trade-off that is involved when introducing virtual functions by comparing the execution time of four