Information and Software Technology 59 (2015) 33–52

Contents lists available at ScienceDirect

Information and Software Technology

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

Automated refactoring to the Null Object design pattern

Maria Anna G. Gaitani, Vassilis E. Zafeiris, N.A. Diamantidis, E.A. Giakoumakis *

Department of Informatics, Athens University of Economics and Business, 76 Patission Str., Athens 104 34, Greece

ARTICLE INFO

Article history:
Received 5 July 2014
Received in revised form 30 October 2014
Accepted 30 October 2014
Available online 8 November 2014

Keywords:
Refactoring
Design patterns
Null Object
Optional fields
Null checks

ABSTRACT

Context: Null-checking conditionals are a straightforward solution against null dereferences. However, their frequent repetition is considered a sign of poor program design, since they introduce source code duplication and complexity that impacts code comprehension and maintenance. The Null Object design pattern enables the replacement of null-checking conditionals with polymorphic method invocations that are bound, at runtime, to either a real object or a *Null Object*.

Objective: This work proposes a novel method for automated refactoring to Null Object that eliminates null-checking conditionals associated with optional class fields, i.e., fields that are not initialized in all class instantiations and, thus, their usage needs to be guarded in order to avoid null dereferences.

Method: We introduce an algorithm for automated discovery of refactoring opportunities to Null Object. Moreover, we specify the source code transformation procedure and an extensive set of refactoring preconditions for safely refactoring an optional field and its associated null-checking conditionals to the Null Object design pattern. The method is implemented as an Eclipse plug-in and is evaluated on a set of open source Java projects.

Results: Several refactoring candidates are discovered in the projects used in the evaluation and their refactoring lead to improvement of the cyclomatic complexity of the affected classes. The successful execution of the projects' test suites, on their refactored versions, provides empirical evidence on the soundness of the proposed source code transformation. Runtime performance results highlight the potential for applying our method to a wide range of project sizes.

Conclusion: Our method automates the elimination of null-checking conditionals through refactoring to the Null Object design pattern. It contributes to improvement of the cyclomatic complexity of classes with optional fields. The runtime processing overhead of applying our method is limited and allows its integration to the programmer's routine code analysis activities.

1. Introduction

Dereferencing null object references leads to runtime errors and, thus, causes program termination or abnormal operation. In order to avoid such errors, the programmer needs to decide which object references can have a null value and introduce appropriate null-checking conditional statements to avoid null dereferences. Compile-time detection of null dereferences is an effective approach for the discovery of many null-related bugs [1] that is not yet integrated in popular languages such as Java or C#. Static code analysis techniques are, also, applied for the discovery of null dereferences (e.g. [2,3]) and are gradually integrated in popular code review tools, such as FindBugs for Java [4,5].

Although null-checking conditionals are a straightforward solution against null dereferences, their frequent repetition is, often, considered a code "smell", i.e. a sign of poor program design. Fowler [6] and Kerievsky [7] document this code "smell", in their books on software refactoring, as a source of code duplication that, also, increases code complexity and, thus, impacts its comprehension and maintenance. Fowler [6] focuses on repeated null checks, scattered in the code of a method or class, that refer to a specific object reference. Kerievsky [7] emphasizes on null checks that involve class fields. Both works suggest the elimination of the null-checking conditionals through refactoring to the Null Object design pattern [8,9]. The Null Object hides the absence of an object (null value) with a substitutable alternative object, namely the *Null Object*, that has the same interface as the real object, but provides a default "do nothing" behavior [9]. The term default "do nothing"

* Corresponding author at: Patission 76, 104 34 Athens, Greece. Tel.: +30 2108203183; fax: +30 2108203275.

E-mail addresses: mariannag17@gmail.com (M.A.G. Gaitani), bzafiris@aueb.gr (V.E. Zafeiris), nad@aueb.gr (N.A. Diamantidis), mgia@aueb.gr (E.A. Giakoumakis).

http://dx.doi.org/10.1016/j.infsof.2014.10.010
0950-5849/

behavior denotes that all methods of the *Null Object* class are implemented so as to either have an empty body or return default results. The lifecycle of a *Null Object* ends with the assignment of a non-null value to the object reference.

The Null Object design pattern enables the replacement of null-checking conditionals with polymorphic method invocations that are bound, at runtime, to either a real object or a *Null Object*. The pattern removes duplicate code fragments that are relevant to (a) null-checks on the same object references and (b) repeated "do nothing" behavior that is executed in the case of a null reference. The latter is extracted to appropriate methods of the *Null Object* class. Besides its contribution to code simplicity, Null Object enables easy and safe program extensions. Specifically, adding method invocations to a potentially null object reference gets simpler and less error prone, since the programmer is not required to remember and introduce relevant null checks. Finally, Null Object increases reusability, as instances of a *Null Object* class can be used in multiple cases of null-checks on the same object type.

This paper deals with the problem of automated refactoring to the Null Object design pattern. It complements the works of Fowler [6] and Kerievsky [7], focusing on the mechanics of the manual application of the refactoring, with a novel method for automated discovery of null-checking conditionals that can be effectively refactored to Null Object. Our analysis focuses on special cases of null-checking conditionals that are encountered in classes with optional collaborators, i.e., with fields that are not always initialized. These conditionals protect optional field dereferences and enclose the behavior of an "empty" collaborator. Moreover, we specify an extensive set of refactoring preconditions that mark cases that can be safely refactored without changing the external behavior of the system. The refactoring identification procedure is complemented with a detailed description of the source code transformation for applying the Null Object design pattern to a given optional field and its respective null-checking conditionals. Our method for automated refactoring to Null Object has been implemented as part of the JDeodorant Eclipse plug-in [10], a tool for the automation of complex Java code refactorings. Moreover, it has been experimentally evaluated on a set of open source Java projects. Several refactoring candidates have been discovered in these projects and their refactoring lead to improvement of the cyclomatic complexity of the affected classes. The successful execution of the projects' test suites, on their refactored versions, provides empirical evidence on the soundness of the proposed source code transformation. Finally, runtime performance results highlight the potential for applying our method to a wide range of project sizes.

The rest of this paper is structured as following: Section 2 presents relevant work on the research area of refactoring to design patterns. Section 3 presents the Null Object design pattern, its alternative implementations and introduces appropriate terminology that will be used in this paper. Section 4 specifies our method for automated identification of refactoring candidates and their elimination through the Null Object design pattern. Section 5 presents an evaluation of this work on the basis of a prototype implementation integrated to the JDeodorant Eclipse plug-in [10]. Finally, Section 6 summarizes the conclusions of this work.

2. Related work

Our work contributes to the research area of automated refactoring to design patterns. Refactoring to patterns aims at the elimination of design flaws through the introduction of appropriate design patterns. The automation of refactoring tasks enables integration of the continuous design improvement practice to the development workflow. This section provides a review on methods for automated refactoring to design patterns. It encompasses approaches relevant to both structural (Abstract Factory, Composite) and behavioral (Decorator, Template Method, State/Strategy) design patterns. As concerning refactoring to Null Object, this work is the first that studies its automation. For an extensive review on the broader research area of software refactoring, the reader may refer to the work of Mens and Tourwe [11].

2.1. Abstract Factory

Refactoring to Abstract Factory is among the earlier approaches on refactoring to patterns. Specifically, Tokuda and Batory [12] proposed the introduction of the design pattern as a composition of parameterized object-oriented transformations. The method provides a specification of these primitive transformations and applies them through appropriate tool support. The introduction of the Abstract Factory is demonstrated through a simple case study.

Refactoring to design patterns is also treated as a series of mini-transformations in the methodology proposed by Cinneide and Nixon [13]. A mini-transformation comprises pre-conditions, post-conditions, transformation steps and an argument over how the mini-transformation supports behavior preservation after its application. The methodology is primarily focused on structure-rich, rather than behavioral patterns, and is applied to refactoring to Abstract Factory.

A logic programming approach to refactoring to patterns has been proposed by Jeon et al. [14]. The method employs logic inferencing for the identification of refactoring opportunities in a Java code base, and the, subsequent, selection of an appropriate strategy for source code transformation. Inference is based on the extraction of system design from Java code and its representation as a set of Prolog-like predicates that are then converted to Prolog facts. Inference rules are, also, specified for each target pattern, that are transformed to Prolog rules. The identification of refactoring opportunities takes place through issuing of Prolog queries. The method applied for the discovery of refactoring candidates to Abstract Factory. The application of the refactoring is based on the mini-transformations approach of Cinneide and Nixon [13].

2.2. Composite

Jebelean et al. [15] use logic metaprogramming for the detection of incorrect applications of the Composite design pattern. The approach involves transformation of the Java project's Abstract Syntax Tree (AST) into Prolog facts through the JTransformer engine [16]. Problematic code fragments are identified through the definition of appropriate Prolog rules. Ajouli et al. [17] focus on the automated transformation of a Visitor pattern instance to Composite and vice versa. The transformation is based on a set of refactoring preconditions that ensure its correct application and reversibility. Moreover, the authors present variations of the base transformation for handling special cases of relaxed preconditions.

2.3. Decorator

Rajesh and Janakiram [18] employ logic programming for the identification of refactoring candidates (or Intent Aspects in the terminology of the paper) to the Decorator design pattern. The method involves construction of the Java project's AST and generation of Prolog facts that reflect its design. Fact generation is based on Predicate Templates, i.e. predefined facts that are introduced to the facts-base during traversal of the project's AST. Moreover, the authors specify Prolog rules for the identification of refactoring candidates to Decorator. The application of the refactoring is enabled through a third party refactoring tool.