ICAI 2023
12th International Conference on Applied Informatics
Eger, Hungary, 2023

Program Code Quality Improvement by using Design Patterns and Decision Merging

Gábor Kusper[ab], Imre Baják[cd], Attila Adamkó[bd], Szabolcs Márien[d]

[a]Eszterházy Károly Catholic University, Faculty of Informatics

[b]University of Debrecen, Faculty of Informatics

[c]Budapest Business School

[d]InnovITech Kft.

Abstract

The aim of this paper is to present techniques about code quality improvement of object-oriented programs when the source code contains redundant decisions. A decision could be an IF statement, a SWITCH statement or a polymorphic method call. Decisions are redundant when we repeat the same predicate check to decide which behavior we shall perform. If we use a polymorphic method call, then there is no predicate check, because the decision is made by late binding, so in this case we have no redundancy. We show how to refactor IF and SWITCH statements by using polymorphic method calls. We call such techniques decision merging in general. Our examples come from the realm of design patterns [3]. First we show a code which contains redundancy, then we apply some design pattern to improve the code. We use the following ones: Adapter, Bridge, Builder, and Decorator. This means that we present a set of before-after examples. This paper is the continuation of [1], which discussed these 4 design patterns: Null Object, Template Method, State, and Strategy.

We believe that these examples are valuable for those research groups which work on automatic refactoring tools. They can use our examples to test their solutions.

The code quality of object-oriented programs is an abstract notion, as a result

it has a different meanings for different programmers. Meanwhile, clean code is a kind of common sense and a well-documented notion [5]. The common sense for OOP programmers is the following: the cleaner the code is, the better its quality is. One of the criteria of clean code is not to repeat yourself, which is also known as the DRY principle.

The other way to approach the notion of code quality, beside clean code, is to use of code quality metrics. The easiest metric is LoC, which means the number of lines in the code. The next one is the cyclomatic complexity (CC) metric, which counts the linearly independent paths of a method. The less the value of CC is, the easier it is to understand the method. Moreover, the less the value of CC is, the easier it is to write unit tests for the method with 100% code coverage.

We improve the code quality by refactoring repeated predicate checks of IF and SWITCH statements, which are usually called selections, or branching statements, but we call them decisions in this paper. A decision selects from the possible behaviors, and the running code performs the selected branch.

So, selection statements, like IF and SWITCH statements, are decisions, but a polymorphic method call is also a decision. In case of polymorphic method calls, say $REF.FOO()$, we have a reference, REF, of type $PARENT$, which points to an object of type $CHILD$, where $CHILD$ is either $PARENT$ itself or one of the child classes of $PARENT$. Because of late binding, the call $REF.FOO()$ calls the method $FOO()$ from $CHILD$ not from $PARENT$. Let us suppose that we have several child classes of $PARENT$. All of them implement the $FOO()$ method. These are the possible behaviors. When we give a value to REF then this is an original decision, which is repeated when we do the polymorphic call $REF.FOO()$. But in this case, we do not need to repeat the predicate check which selects the proper behavior, instead of that, the running framework performs a late binding. So, there is no free lunch!

So, in case of polymorphic method calls we can distinguish between the original decision and the repeated decisions. Can we do the same in case of IF and SWITCH statements? Without loss of generality we study IF - ELSE IF structures. Note that a SWITCH statement can be transformed into an IF - ELSE IF structure. The IF statement has the syntax: IF (predicate) BLOCK1 [ELSE BLOCK2], where the [ELSE BLOCK2] part is optional. Its semantics is the following: if the predicate evaluates to true, then BLOCK1 is executed, otherwise, if there is an ELSE part, BLOCK2 is executed. BLOCK2 can also be an IF statement. In this way we can build an IF - ELSE IF structure.

Back to the previous question, what are the original decisions in case of an IF statement? In case of an IF statement, the original decisions are those assignments where we assign a value to those variables which are involved in its predicate.

For example, if we have the statement: IF (X == 1) BLOCK1, then the original decision is the assignment, where X gets its value, like: X = 1. And the repeated decisions are those IF statements which check the value of X in their predicate, like: IF (X == 1) BLOCK1.

In this paper we give examples of how to refactor repeated predicate checks

by using polymorphic method calls. If we do so, then we get a better CC value, and also other code quality metrics become better, like REDC1, REDC2, MDA1, MDA2 [4].

We select examples from the realm of design patterns. First, we present an ugly code which is in "before applying the pattern" state. Then we present a nicer code, an instance of a design pattern, so it is in "after applying the pattern" state.

Design patterns are well-designed, reusable solutions for common programming tasks. One of the first design patterns was MVC (Model – View – Controller). Design patterns can be seen as a lower level abstraction compared to Design Principles [2].

There is a trend that forces programmers to give up their freedom. We cannot modify the memory anymore as easily as in the C programming language; we have no pointers; we cannot retrieve the memory address of a variable. We cannot do that because those techniques are dangerous, it is easy to introduce an error using them. We have this trend, because programming is not an art anymore, but a normal job, where there are masters and a lot of youngsters. It is not a good idea to give them dangerous tools.

The next logical step in this trend is to force programmers to use design patterns. In a few years there will be tools which warn programmers that they could introduce design patterns, and even do the refactoring. These tools need a lot of examples for codes before introducing the design pattern and after that. With the help of such examples research groups can test and teach their solutions. It seems that there are very few works in this field, therefore, we have decided to share some examples we have created.

Each example has a common structure: we have a Java code which contains some redundancy. This is the code before introducing a design pattern. The next version has the same behavior, i.e., the same functionality, but contains no redundancy. This is the code after introducing a design pattern.

In this paper, we discuss the following design patterns: Adapter, Bridge, Builder, and Decorator.

We believe that our before-after examples are valuable for those research groups which work on automatic refactoring tools.

Finally, we use some code quality metrics to measure the examples. The selected metrics are: REDC1, REDC2, MDA1, MDA2, and CC.

References

[1] M. Danisovszky, A. Adamkó, I. Baják, K. Kusper, S. Márien, G. Kusper: *Cognitive Code Quality Improvement with Pattern Recognition and Recommendation by Examples*, in: CogInfoCom, IEEE, 2020, doi: 10.1109/CogInfoCom50765.2020.9237896.

[2] M. Danisovszky, T. Nagy, K. Répás, G. Kusper: *Western Canon of Software Engineering: The Abstract Principles*, in: CogInfoCom, IEEE, 2019, doi: 10.1109/CogInfoCom47531.2019.9089999.

[3] E. Gamma, R. Helm, R. E. Johnson: *Design Patterns. Elements of Reusable Object-Oriented Software*. 1st ed., Reprint., Addison-Wesley Longman, Amsterdam, 1994, ISBN: 0201633612, URL: http://www.amazon.de/Patterns-Elements-Reusable-Object-Oriented-Software/dp/0201633612/ref=sr_1_1?ie=UTF8&qid=1302724786&sr=8-1.

[4] S. Márien: *Decision structure based object-oriented design principles*, Annales Mathematicae et Informaticae 47 (2017), pp. 149–176.

[5] R. C. Martin: *Clean Code: A Handbook of Agile Software Craftsmanship*, Robert C. Martin Series, Upper Saddle River, NJ: Prentice Hall, 2008, ISBN: 978-0-13235-088-4, URL: https://www.safaribooksonline.com/library/view/clean-code/9780136083238/.