Towards an Investigation of Opportunities for Refactoring to Design Patterns

Norihiro Yoshida, Katsuro Inoue
Graduate School of Information Science and Technology, Osaka University
1-3, Machikaneyama-cho, Toyonaka, Osaka, 560-8531, Japan
{n-yosida, inoue}@ist.osaka-u.ac.jp

Abstract

Refactoring is a well-known process to improve the maintainability of object-oriented software. Recently, it is said that refactoring to design patterns can improve design quality of maintaining software. However, there are a few case studies of refactoring to design patterns. This position paper shows our approach to investigate opportunities for refactoring to design patterns in software systems. In our approach, we will develop an automated tool that identifies opportunities for refactoring to design patterns, and then will carry out the investigation using our tool.

1 Introduction

Motivation *Refactoring*[2] is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. That is to say, refactoring is a process to improve the maintainability of software systems. Recently, it is said that refactoring to *design patterns*[3] can improve design quality of maintaining software systems that have lack of application of design patterns[5][7]. However, since there are a few practical case studies of refactoring to design patterns[7], it is not clear what kind of refactoring opportunities there are in software systems. Therefore, we are planning to investigate opportunities for refactoring to design patterns in software. And for that purpose we are also planning to develop an automated tool that identifies opportunities for refactoring to design patterns.

An Example of Refactoring to Patterns J. Kerievsky made a catalogue of refactorings to patterns[5]. His catalogue includes 27 pairs of a description of a refactoring opportunity and the corresponding procedure for performing refactorings using a design pattern. Here, as an example, we explain the refactoring is called *Introduce Polymorphic Creation with Factory Method* described in his catalogue. The refactoring opportunity in his catalogue is defined as *"Classes in a hierarchy implement a method similarly except for an object creation step"*. Similar method can be called *code clone*[4] or *duplicated code*[2]. *Code clone* is generally considered as one of factors that make software maintenance more difficult[4]. Figure 1 shows an example of the refactoring described in his catalogue. As shown in Figure 1(a), the targets of the refactoring are test classes DOMBuilderTest and XMLBuilderTest for DOMBuilder and XMLBuilder, respectively. Because target classes have similar methods except for an object creation step, they imply the opportunity for *Introduce Polymorphic Creation with Factory Method*. This refactoring is comprised of two steps. As shown in Figure 1(b), first, a common superclass (AbstractBuilderTest) for the target classes is introduced, and similar methods in the target classes are merged into new method in the common superclass. Second, a *Factory Method*[3] is introduced in each of the common superclass (AbstractBuilderTest) and the subclasses (DOMBuilderTest and XMLBuilderTest). *Factory Method* means a method is primarily intended to create an object. Because of removing code duplication and introducing *Factory Method*, it is easier to add new test class as a subclass of AbstractBuilderTest than before.

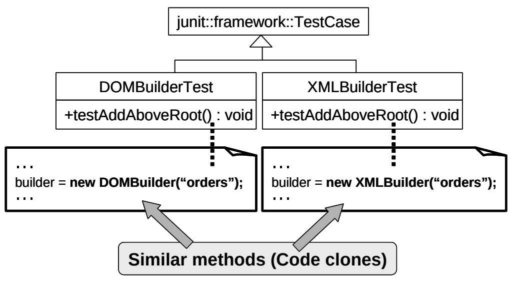
2 Investigation Plan

Automation of Identifying Opportunities For our investigation, we will develop the automated tool that identifies opportunities for refactoring to design patterns. To identify codes shown in Figure 1(a), the automated method has to find codes that satisfy the following conditions:

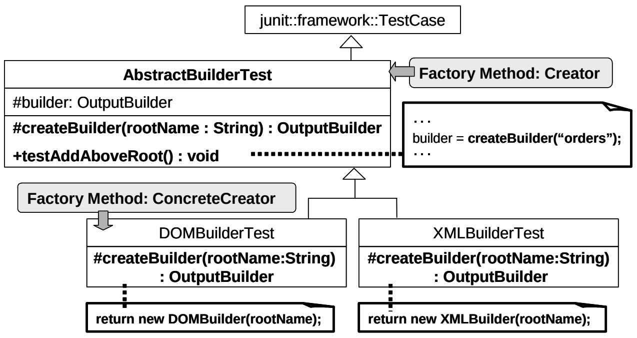
C1 Similar methods belong to classes have common parent classes

C2 Only difference among similar methods is an object creation step

Figure 1 is a special case of *Form Template Method Refactoring*[2], thus we presented C1. C2 is presented for introducing *Factory Method*. We are planning to judge those conditions by the steps below.


junit::framework::TestCase
DOMBuilderTest
+testAddAboveRoot() : void
XMLBuilderTest
+testAddAboveRoot() : void
...
builder = new DOMBuilder("orders");
...
...
builder = new XMLBuilder("orders");
...
Similar methods (Code clones)

(a) Before refactoring


junit::framework::TestCase
AbstractBuilderTest
#builder: OutputBuilder
#createBuilder(rootName : String) : OutputBuilder
+testAddAboveRoot() : void
Factory Method: Creator
...
builder = createBuilder("orders");
...
Factory Method: ConcreteCreator
DOMBuilderTest
#createBuilder(rootName:String) : OutputBuilder
XMLBuilderTest
#createBuilder(rootName:String) : OutputBuilder
return new DOMBuilder(rootName);
return new XMLBuilder(rootName);

(b) After refactoring

Figure 1. Introduce Polymorphic Creation with Factory Method[5]

Step1 Detect similar methods using a code clone detection tool such as CCFinder[4].

Step2 Evaluate whether detected methods belong to classes that have common superclasses and whether they include object creation statements.

Except that 4 kinds of opportunities do not correspond to refactoring that introduces design pattern, J. Kerievsky's book includes 23 refactoring opportunities. Also, it is able to identify the other 3 kinds of opportunities based on code clone detection and syntactic analysis. They are *Replace Conditional Logic with Strategy*, *Form Template Method*, *Replace One/Many Distinctions with Composite*, and *Introduce Null Object*. Otherwise, because the descriptions of other 3 kinds of opportunities include existence of design pattern instances, it is necessary to use design detection methods[8] for identifying those opportunities. They are *Encapsulate Composite with Builder*, *Extract Composite*, and *Extract Adapter*. For the rest of kinds of opportunities, we have to need further discussion. The novelty of our tool is the identification based on code clone and the opportunities proposed by J. Kerievsky.

Investigation points In the investigation, we will focus on the following points:

- How many refactoring opportunities exist in OSS (Open Source Software) written in Java?
- Is it possible to improve maintainability by performing refactoring to design patterns?

For evaluating whether maintainability is improved, we are planning to use *CK Metrics*[1] and *Design principles*[6]. We use them to evaluate software quality by both quantitative and qualitative criteria.

Anticipated Results We believe that there are differences among design patterns about a number of refactoring opportunities and how much maintainability is improved. In addition, from refactoring perspective, we expect to discuss the quality of each design patterns based on the result of the investigation.

3 Concluding Remarks

In this paper, we showed a plan for investigating opportunities for refactoring to design patterns in OSS. First, we introduced an example of refactoring to design patterns. Then, we described an automated tool that identifies opportunities for refactoring to design patterns. Finally, we explained our investigation points and anticipated results.

Acknowledgment This work is being conducted as a part of Stage Project, the Development of Next Generation IT Infrastructure, supported by MEXT.

References

[1] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
[2] M. Fowler. *Refactoring: improving the design of existing code*. Addison Wesley, 1999.
[3] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
[4] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.
[5] J. Kerievsky. *Refactoring to Patterns*. Addison Wesley, 2004.
[6] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Printice Hall, 2003.
[7] J. Rajesh and D. Janakiram. JIAD: A tool to infer design patterns in refactoring. In *Proc. of PPDP 2004*, pages 227–237, 2004.
[8] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, 2006.