Assessing the Quality of Refactoring Patterns for Introducing Design Patterns

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

ABSTRACT

Refactoring is a well-known process to improve the code design of object-oriented programs. Recently, several literatures focus on refactoring with introducing design patterns that are general repeated solutions to common problems in software design. For making it easy to perform such refactoring, a lot of refactoring patterns are proposed. Each refactoring pattern includes a description of refactoring opportunities (i.e., when a software system should be performed refactoring with introducing design patterns) and the corresponding procedure (i.e., how to perform refactoring with introducing design patterns). However, the usefulness of each refactoring pattern is not clearly assessed. This paper describes our approach to assess the quality of refactoring patterns for introducing design patterns and also shows the assessment of open source software.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maitenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Patterns*

General Terms

Design, Languages, Experimentation

Keywords

Quality assessment, Refactoring pattern, Design pattern

1. INTRODUCTION

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. Recently, several literatures focus on refactoring with introducing *Design Patterns* (DPs)[3] that are general repeated solutions to common problems in software design. Refactoring with introducing DPs can improve design quality of maintaining software systems that lack of application of DPs[5, 8]. For making it easy to perform such refactoring, Kerievsky proposed 27 refactoring patterns[5]. Each refactoring pattern includes a description of refactoring opportunities (i.e., when a software system should be performed refactoring with introducing DPs) and the corresponding procedure (i.e., how to perform refactoring with introducing DPs).

However, the quality of each refactoring pattern is not clearly assessed. We can not determine which refactoring patterns should be given priority to apply. Also, we do not know whether it is easy to apply each refactoring pattern.

P1 How many refactoring opportunities are involved in software systems?

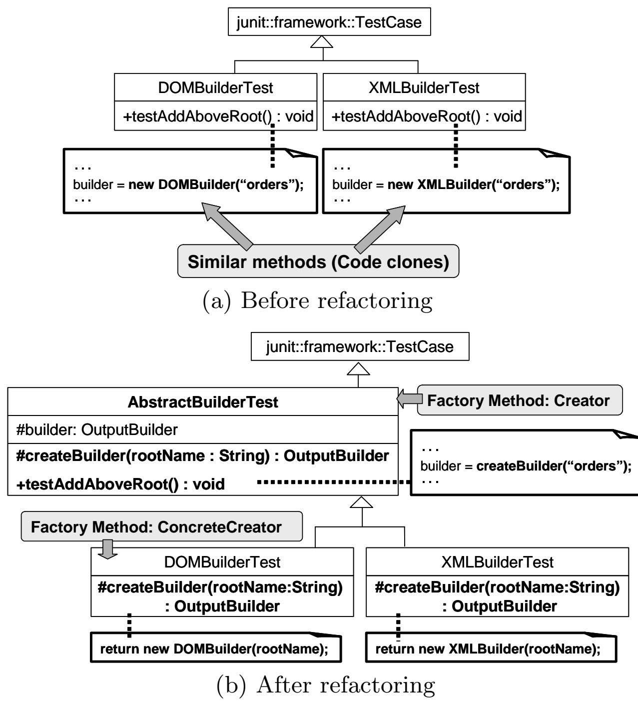
P2 Is it easy to perform refactoring to those opportunities?

In this paper, we propose an approach based on above points to assess the quality of refactoring patterns introducing DPs. First of all, we briefly explain about refactoring with introducing DPs. Then we propose a plan for assessing the quality of such refactoring patterns and present an automated tool that identifies opportunities for applying one of the refactoring patterns described in Kerievsky's book[5]. Finally, we show a case study of *Open Source Software* (OSS) and discuss P1 and P2. In the case study, we identify such opportunities in OSS by using that automated tool and perform refactoring to those opportunities.

2. REFACTORING WITH INTRODUCING DESIGN PATTERNS

Kerievsky made a catalogue of refactorings introducing *Design Patterns* (DPs)[5]. His catalogue includes 27 pairs of a description of a refactoring opportunity and the corresponding procedure for performing refactorings introducing a DP.

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*[1, 4] or *duplicated code*[2]. *Code clone* is generally considered as one of factors that make software maintenance more difficult[1, 4].



(a) Before refactoring

(b) After refactoring

Figure 1: Introduce Polymorphic Creation with Factory Method

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 following two steps.

Step1 As shown in Figure 1(b), 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.

Step2 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.

3. ASSESSMENT PLAN

The aim of our plan is to assess the quality of each refactoring pattern introducing *Design Patterns* (DPs). In our plan, we assess the following points for each refactoring pattern.

- **Number of Refactoring Opportunities:** The number of refactoring opportunities in a software system is closely related to the usefulness of a refactoring pattern for the system. Therefore, we assess the number of refactoring opportunities in a software system. The assessment requires an automated tool that identifies opportunities for refactoring. For our case study, we have developed the tool that identifies opportunities for the refactoring described in section 2. The detail of the tool is introduced in section 4.

- **Ease of Refactoring:** The major purpose of refactoring is to reduce maintenance cost[2]. Therefore, if it is too costly to apply a refactoring pattern, it is not desirable to perform the refactoring[6]. In our plan, we assess the ease of applying each refactoring pattern. For each performed refactoring, we confirm the performed procedures other than which are described in Kerievsky's book[5].

4. AUTOMATION OF IDENTIFYING OPPORTUNITIES

For our case study, we have developed the tool that identifies opportunities for *Introduce Polymorphic Creation with Factory Method* described in section 2.

According to Kerievsky's book, to identify code shown in Figure 1(a), the automated method has to find code that satisfies 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*[3]. The automated tool judges those conditions by the steps below.

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

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

5. CASE STUDY

We present a case study as an example of implementing our plan. In our case study, we identify *Introduce Polymorphic Creation with Factory Method* in *Open Source Software* by using the tool described in section 4, and then perform refactoring to those opportunities. The target software systems are ANTLR, Ant and Azureus which are written in Java.

Table 1 shows the numbers of refactoring opportunities in each target system. We could confirm that a system, which is comprised of relatively large number of classes, has a tendency to involve a lot of opportunities.

Figure 2 shows the performed refactoring in ANTLR. This refactoring needs for additional procedure other than that

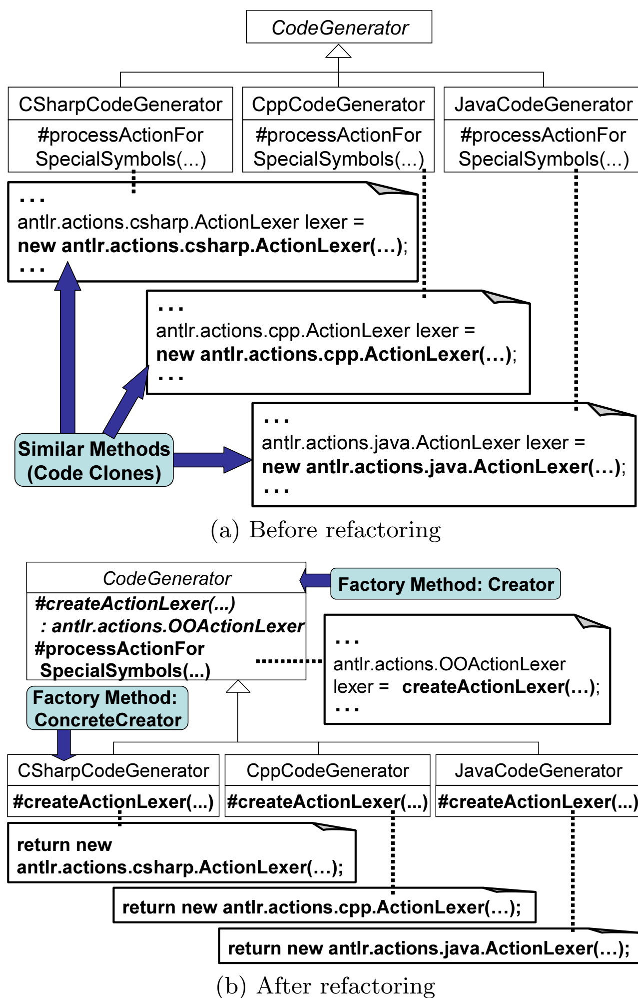


Figure 2: Refactoring in ANTLR

describe in Kerievsky's book[5]. In Figure 2(a), three types of the created objects (i.e. antlr.action.csharp.ActionLexer, antlr.action.cpp.ActionLexer and antlr.action.java.ActionLexer) do not have any common superclass or superinterface in target source code. In this situation, we can not introduce a *Factory Method* in each class because it is not able to determine appropriate return type of each *Factory Method*. To break this situation, we created the antlr.action.OOActionLexer interface as the common superinterface of those types. Then we introduced a *Factory Method* (i.e. createActionLexer) that returns antlr.action.OOActionLexer in each class (Figure 2(b)).

6. DISCUSSION

Table 1: The target software systems

Name	LOC	#classes	#opportunities
ANTLR 2.7.4	47K	285	1
Ant 1.7.0	20K	778	2
Azureus 3.0.3.4	538K	2226	14

Each target system has at least one opportunity for *Introduce Polymorphic Creation with Factory Method* refactoring. Especially, Azureus has 14 opportunities.

On the other hand, the refactoring is shown in Figure 2 needs for the introduction of new interface other than that describe in Kerievsky's book[5]. That is to say, we confirmed this refactoring pattern has low ease of refactoring if types of the created objects do not have any common superclass or superinterface in target source code.

7. CONCLUSION

In this paper, we proposed an approach based on above points to assess the quality of refactoring patterns introducing DPs. We proposed a plan for assessing the quality of such refactoring patterns and presented an automated tool that identifies opportunities for applying the *Introduce Polymorphic Creation with Factory Method* pattern. Also, we showed a case study of ANTLR, ANT and Azureus, and then discussed the quality of the pattern regarding the number of refactoring opportunities and the ease of refactoring.

We are planning to assess the other refactoring patterns, such as *Replace One/Many Distinctions with Composite* and *Encapsulate Composite with Builder*. We believe that it is able to detect several opportunities for applying the refactoring patterns based on code clone detection or design pattern detection [7, 9].

Acknowledgments

This reseach was supported by JSPS, Grant-in-Aid for Scientific Research (A) (No.17200001).

8. REFERENCES

[1] B. S. Baker. Finding clones with Dup: Analysis of an experiment. *IEEE Trans. Sofw. Eng.*, 33(9):608–621, 2007.

[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 multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Sofw. Eng.*, 28(7):654–670, 2002.

[5] J. Kerievsky. *Refactoring to Patterns*. Addison Wesley, 2004.

[6] R. Leitch and E. Stroulia. Assessing the maintainability benefits of design restructuring using dependency analysis. In *Proc. of METRICS 2003*, pages 309–322, 2003.

[7] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. of ICSE 2002*, pages 338–348, 2002.

[8] J. Rajesh and D. Janakiram. JIAD: A tool to infer design patterns in refactoring. In *Proc. of PPDP 2004*, pages 227–237, 2004.

[9] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design pattern detection using similarity scoring. *IEEE Trans. Sofw. Eng.*, 32(11):896–909, 2006.