Refactoring Formal Specifications in Object-Z

Hui Liu
School of Computer Science and Technology
Beijing Institute of Technology
Beijing 100871, P.R.China

Bin Zhu
School of Computer and Information
Shanghai Second Polytechnic University
Shanghai, 201209, P.R.China

Abstract

Software refactoring is to restructure artifacts to improve software quality, especially readability, extensibility, and maintainability, while preserving its external behaviors. Software refactoring has been successfully applied to source code and design models. However, refactoring has not yet been introduced to formal specifications. Compared to source code in programming languages similar to nature languages, formal specifications in formal mathematic languages are in urgent need of refactoring because mathematic languages are usually more difficult to understand or modify. Furthermore, formal specifications' inherent formality makes it easier to validate behavior preservation of refactorings, which dramatically increases the dependability of refactorings. This paper illustrates the necessity and possibility of refactoring formal specifications. It first illustrates the necessity with an motivating example, and then proposes a series of primitive refactorings and advanced refactorings that are composed of primitive refactorings.

Keywords: *Software Refactoring, Formal Specifications, Z, Object-Z*

1 Introduction

Formal specifications here mean software specifications in formal specification languages, such as Z [2] and Object-Z [6]. Since formal specifications can be reasoned, proved and transformed automatically into executable code, they are treated as a promising approach for the development of mission-critical, safety-critical and security-critical systems. As a result, formal specifications are gaining their deserved acknowledge in both the literature and industry.

However, formal specifications are usually a bit hard to read, let along extend. Though we have gained great progresses in development of compilers for formal specification languages, it is still necessary to read and extend formal specifications manually in order to adapt them into new environments and new requirements. So, technologies are urgently needed to increase their readability and extensibility. And refactoring is an excellent candidate to do this job.

Formal specifications need refactoring to make them easier to understand and extend. From the above discussion, it is easy to find the following fact. On the one hand, it is hard to understand and extend existing formal specifications. However, these actions are inevitable because of software evolution. On the other hand, refactoring is a technology to make things easier to understand and extend without changing their external behaviors. So, it is nature to apply refactoring to formal specifications.

Formal specifications is more suitable for refactoring than other informal artifacts. One focus of refactoring research is how to prove that a refactoring is behavior-preserving which is a key property of refactoring. Up to date, the main technology for this issue is formalisms [3]. For example, Roberts [5] describes pre-conditions of refactoring with first order predicate calculus. Ward and Bennett [8] take WSL in stead. However, it is much easier for formal specification refactoring because formal specification languages themselves are powerful enough to specify pre-conditions and post-conditions of refactorings.

In this paper, we will take Object-Z as an example for refactoring. Z is one of the most popular formal specification languages. In order to introduce object-oriented properties into Z, Object-Z is invented [1][6]. We choose them for refactoring not only because of their representativeness but also rich tool support which would probably lead to automation of some refactorings. Tool support can increase efficiency dramatically[7].

2 Refactorings of Formal Specifications in Object-Z

2.1 Primitive Refactorings

In this subsection, we will present some primitive refactorings for specifications in Object-Z. Primitive refactorings

978-0-7695-3336-0/08 $25.00 © 2008 IEEE
DOI 10.1109/CSSE.2008.260

are usually simple and intuitive. They are foundations of advanced refactorings listed in the next subsection 2.2.

2.1.1 Renaming Entities

Entities (classes, variables, operations, parameters et al.) can be renamed with more intuitive names. For example, $Queue[ElementType]$ is much better than $Class1[P1]$. Since renaming elements is a usual technology used in Object-Z, there is no need to prove its behavior-preserving property.

2.1.2 Adding New Entities

Unreferred new classes, variables and operations would not change the behavior of the resulting system. So, the refactoring is behavior-preserving. It is usually used with other refactorings, such as *Replacing Expressions with Equivalent Expressions*. It makes no sense to apply it in isolation.

2.1.3 Removing Entities

Removing unreferred classes (except the one representing the system), variables and operations would not change the behavior of the resulting system too. Furthermore, if a predicate is s tautology. then it can be removed from corresponding predicate list (of axiomatics, state schemas or operation schemas) without changing its semantics. For example, in the following schema, t is a tautology. When it is removed, the resulting predicate list is still semantically equivalent to the original one.

$$x = \neg\ y$$
$$t = x \vee y$$

By checking the whole specification (tree traversal), unreferred entities can be found automatically. And it is also possible to detect tautologies automatically. So, removing entities can be automated.

2.1.4 Replacing Expressions with Equivalent Expressions

For example, if a state invariant or axiomatic of a class says that $x = y \wedge g \wedge f$, we can simplify expression $z \wedge y \wedge g \wedge f \wedge m$ within the class into $z \wedge x \wedge m$ with replacement refactoring. It simplifies the specification and thus increases its readability. Usually, this is used with other refactorings, such as *Adding New Entities* (variables or axiomatics).

2.1.5 Moving Entities

Object-Z gives rich rules to guide this kind of refactoring. For example, expression 1 and expression 2 are obtained by moving subexpressions within $f(x) \wedge g(y) \wedge h(x) \wedge k(y)$. The three expressions look different, but in fact they are equivalent.

$$f(x) \wedge h(x) \wedge g(y) \wedge k(y) \quad (1)$$
$$g(y) \wedge k(y) \wedge f(x) \wedge h(x) \quad (2)$$

Refactoring of *Moving Entities* is usually combined with other refactorings to form a more complicated and meaningful refactoring, such as *Separating Qualifiers* and *Separating Pre-Conditions from Post-Conditions* which will be illustrated later.

2.2 Advanced Refactorings

Primitive refactorings can be combined together to form a more complicated and powerful refactoring. In this subsection, we will give some representative advanced refactorings for specifications in Object-Z.

2.2.1 Refactoring with Generic Entities

Generic classes are helpful to eliminate duplicate. It also makes specifications easier to extend. Generic schemas do the same job for specifications in Z. In order to combine two or more entities (classes or schemas) into a generic entity, we should make sure that they share similar structures and functions. Suppose a retype transition is to retype a variable or parameter. If we are going to apply this refactoring to schemas (or classes) s_1 and s_2, there must be a sequence of such transitions to transform s_1 into s_1' which is semantically equivalent to s_2. However, it is just a necessary condition: it is not sufficient anyway. Before applying this refactoring, we have to consider the semantics of the new generic entities avoiding to decrease specifications' readability. According to the state of the art, it remains a challenge to automate this detection. But it is possible to provide semi-automatic tools support for this refactoring.

2.2.2 Refactoring with Generalization

Object-Z is an object-oriented language, which makes it possible to apply refactorings for object-oriented programming languages (such as C++ and Java) to Object-Z. One of the most attractive character of object-oriented languages is inheritance. A well-formed inheritance hierarchy is a key to good quality of specifications as well as that of source code. Abstracting common supper classes helps to organize existing concepts (classes), and thus makes specifications easier to understand and extend. Furthermore, abstracting common super classes is also an efficient means for reuse (and thus helps to remove duplicate). In Fig.1, we can see that $Student$ and $Teacher$ are kindred concepts, and thus share

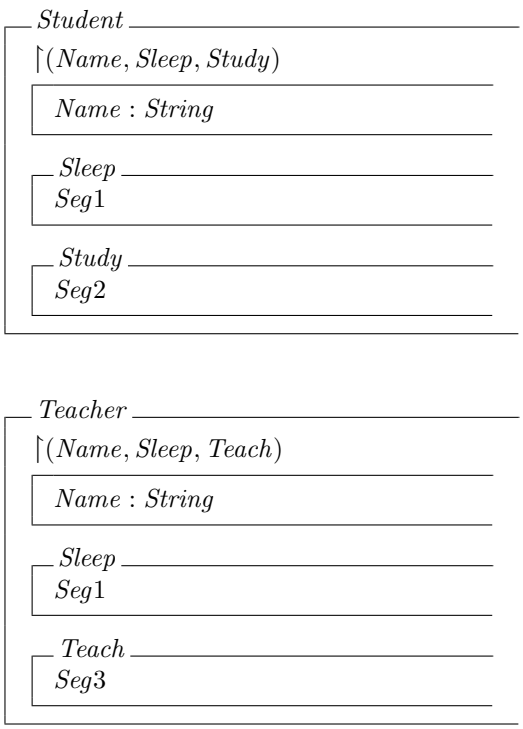

Figure 1. Common Parts in Different Classes

a lot of common variables and operations. With *Refactoring with Generalization*, a super class $Person$ is brought forward to interrelate the two kindred concepts as shown in Fig.2, which draws duplicate out (into the supper class). The refactoring also does some help to evolution of the specification. For example, if librarians are taken into consideration in the next version, it is easy to add a new class $Librarian$ inheriting common super class $Person$. Otherwise, we have to create such a class from scratch.

2.2.3 Refactoring with Specialization

Refactoring with Specialization is in the opposite direction to that of *Refactoring with Generalization*. But in fact, properly used, both of them are efficient approaches for concept organization. As suggested by Opdyke [4], if a complex class embodies both a general abstraction and several different concrete cases, we had better decompose it into several smaller classes. For example, suppose the class $Person$ in Fig.2 contains all features of $Student$ and $Teacher$ instead of only common parts of them. In order to distinguish students from teachers, a special indicator $PersonKind$ has to be added.

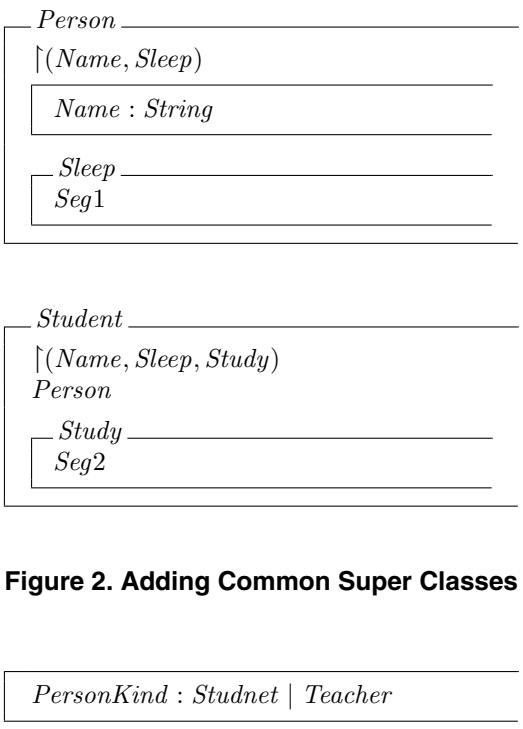

Figure 2. Adding Common Super Classes

When invoking $Teach$ on an instance of $Person$, we have to make sure that it is a teacher by a predicate:

$$PersonKind = Teacher \tag{3}$$

As it shows, such a complex class is confusing and hard to use. As a refactoring with specification, we will split it into three classes: a supper class ($Person$) containing common features and subclasses ($Student$ and $Teacher$) containing special features for each value of $PersonKind$ respectively. And then, objects whose type is $Person$ should be retyped as $Student$ or $Teacher$ except those aiming at polymorphism where the supper class acts as a common interface. The variable $PersonKind$ is removed.

2.2.4 Separating Qualifiers

$\forall\, x\ \forall\, y\,(P(x)\ \wedge\ G(y)\,)$ is equivalent to $(\forall\, x\, P(x)) \wedge (\forall\, y\, G(y)\,)$, but the latter is more intuitive and thus easier to understand. It is quite hard, but still possible to automate this refactoring.

2.2.5 Simplifying Expressions

Since expressions are the main body of a specification, simplifying expressions is also simplifying the specification. One of this kind of effort is to remove tautologies from expressions where the tautologies are connected with others by AND. Of course, contradictions can be remove when they are connected with others by OR.

Another way to this simplification is expression reuse. If a complex subexpression is used by many more complicated expressions, we add a secondary variable [6] and define it with a corresponding invariant. And then, we replace every occurrence of the subexpression within the class with the secondary variable. As a result, expressions become compacter and simpler. Expression reuse is composed of two kinds of primitive refactorings: *Adding New Entities* and *Replacing Expressions with Equivalent Expressions*. So, the property of behavior-preserving is guaranteed. Of course, other kinds of simplification are also available, such as *Separating Qualifiers*. In fact any change to an expression will do some help as long as the following two conditions are guaranteed. First, the resulting expression is equivalent to the original. Second, the resulting expression is easier to understand and extend.

2.2.6 Explaining Literal

Numeric and string literals are usually used without explanations. For example, a numeric literal is used without explanations in the following expression:

$$count < 100 \tag{4}$$

The authors know that the literal 100 presents the max amount of elements of the queue. However, other readers would have difficulty in anticipating its meaning. If we can add a constant variable $MaxAccount = 100$ and replace all the appearances of literal 100 with $MaxAccount$, the specification will be more intuitive.

$$\begin{array}{|l} MaxAccount : \mathbb{N} \\ \hline MaxAccount = 100 \end{array}$$

It also eases changes to the constant: we do not have to find out every occurrence of 100, and check whether it represents the max amount of elements or not, when the max amount is extended to 200. We simply replace $MaxAccount = 100$ with $MaxAccount = 200$.

2.2.7 Simplifying Schemas

Schema decorations Δ and Ξ are excellent tools for refactoring. As shown in Fig. 3, $GetStudentNameA$ is equivalent to $GetStudentNameB$. But, the latter is much simpler and more intuitive. The key of this simplification is $\Xi Student$. This refactoring is quite simple and easy to be automated.

$$\begin{array}{|l} GetStudentNameA \\ \hline Student, Student' \\ ReturnName! : String \\ \hline Name = Name' \\ ReturnName! = Name \\ \hline \end{array}$$

$$\begin{array}{|l} GetStudentNameB \\ \hline \Xi Student \\ ReturnName! : String \\ \hline ReturnName! = Name \\ \hline \end{array}$$

Figure 3. Simplifying Schemas

3 Conclusion

Formal specifications need refactoring to increase their readability and extensibility. Their inherent formality is also a distinct advantage for refactoring automation. We first argue for the necessity of refactoring formal specifications. And then, we explore Object-Z and bring forward a list of refactorings.

References

[1] Roger Duke, Paul King, Gordon Rose and Graeme Smith. The Object-Z Specification Language. Technical Report 91-1, Software Verification Research Centre, Department of Computer Science, University of Queensland, May 1991.

[2] J.M. Spivey. The Z Notation: A Reference Manual, second edition, London, Prentice Hall, 1990.

[3] Tom Mens and Tom Tourwe. A Survey of Software Refactoring. IEEE Transitions on Software Engineering, vol.30, no.2, pp. 126-139, 2004.

[4] W.F Opdyke. Refactoring Object-Oritented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[5] D.Roerts. Practical Analysis for Refactroing. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[6] Graeme Smith. The Object-Z Specification Language. Kluwer Academic Publishers, 2000.

[7] L.Tokuda and D.S.Batory. Evolving Object-Oritented Designs with Refactorings. Automated Software Eng., vol.8, no.1, pp. 89-120, 2001.

[8] M.P.Ward and K.H.Bennett. Formal Mehods to Aid the Evolution of Software. Int'l J. Software Eng. and Knowledge Eng., vol.5, no.1,pp.25-47,1995.