Frame-Based Behavior Preservation in Refactoring

Katsuhisa Maruyama*, Shinpei Hayashi†, Norihiro Yoshida‡ and Eunjong Choi§
* Department of Computer Science, Ritsumeikan University, Japan, maru@cs.ritsumei.ac.jp
† Department of Computer Science, Tokyo Institute of Technology, Japan, hayashi@se.cs.titech.ac.jp
‡ Center for Embedded Computing Systems, Nagoya University, Japan, yoshida@ertl.jp
§ Graduate School of Information Science, Nara Institute of Science and Technology, Japan, choi@is.naist.jp

***Abstract*—Behavior preservation often bothers programmers in refactoring. This poster paper proposes a new approach that tames the behavior preservation by introducing the concept of a frame. A frame in refactoring defines stakeholder's individual concerns about the refactored code. Frame-based refactoring preserves the observable behavior within a particular frame. Therefore, it helps programmers distinguish the behavioral changes that they should observe from those that they can ignore.**

I. Introduction

Behavior preservation is a critical part of the definition of refactoring [2]. Many programmers learned that refactoring improves the design of existing code without changing its observable (or external) behavior [1]. Meanwhile, they know that only the application of behavior-preserving transformations cannot improve existing code [3], [5], [7]. For them, refactoring provides expedient transformations to improve their code.

In this situation, may refactoring violate behavior preservation? The programmers who want to exactly understand past changes cannot afford to overlook this violation since they must check the possibility of behavioral changes that result from possibly unsafe refactorings. Moreover, they prefer to explicitly record a change as a refactoring if it obviously preserves the behavior of the changed code since this recording can help future understanding of the code. Nevertheless, even they sometimes and careless programmers often will ignore behavior preservation in refactoring. It is time to come up with a systematic way to relax the definition of behavior preservation.

II. Motivating Example

To reach consensus on what is behavior preservation, we first describe its definition. The two versions of a program before and after refactoring must produce semantically equivalent results. In other words, if those programs are executed with the same set of input values, the resulting set of output values must be the same [6]. However, since it is hard to formally prove the behavior preservation for every input-output pair based on this definition, a pragmatic way based on a rigorous testing discipline is popular in realistic development [4]. If a sufficient number of test results are equivalent before and after refactoring, a programmer can believe that the applied refactoring preserves the program behavior.

Under the above definition, let us show a code change that complicates the understanding of behavior preservation.

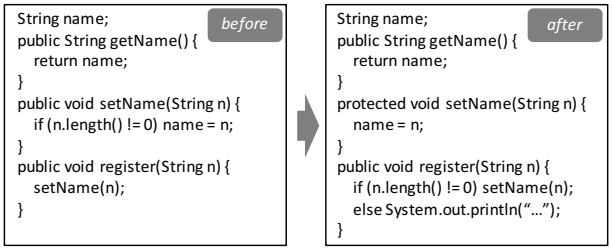

Fig. 1. Code snippets before and after the change.

#	test code	before	after	same?
T1	name = "X"; register("A"); assertEquals("A", getName());	*true*	*true*	***Yes***
T2	name = "X"; register(""); assertEquals("X", getName());	*true*	*true*	***Yes***
T3	name = "X"; setName("A"); assertEquals("A", getName());	*true*	*true*	***Yes***
T4	name = "X"; setName(""); assertEquals("X", getName());	*true*	*false*	***No***

Fig. 2. Test results of the code shown in Figure 1.

Figure 1 presents code snippets within the same class of a program before and after the change. To verify whether this change preserves the behavior of the program, we could prepare the four test cases (whose amount is not sufficient in fact) shown in Figure 2. Strictly speaking, this change alters the program behavior since the two test results of T4 differs. However, not all test cases correspond to the behavior of a program that satisfies its requirements specification and some of them are sometimes designed to just verify implementation of the program.

If a programmer who made this change knows that no method within the program can directly invoke method `setName()` except for method `register()`, and expects this restriction to remain by changing the visibility of `setName()` into "protected", this programmer might pay no attention to the difference between the test results of T4. Especially, if she considers that all application developers may simply use `register()` and not be able to observe this difference, she is likely to affirm that the change is a refactoring. On the other hand, another programmer (who did not make the change) has a high likelihood to judge the change as a non-refactoring since a library developer may write new code that accidentally invokes `setName()`. As a result, the same change could involve programmers with

978-1-5090-5501-2/17/$31.00 © 2017 IEEE

different standpoints. Some programmers ignore the behavioral change of `setName()`, which is revealed by the results of T4, and the other programmers consider the behavior preservation of `setName()` to be important.

III. Approach and Future Direction

Our basic idea assumes that the requirement of behavior preservation depends on the stakeholders who are concerned with the changed code. This means that the observable behavior has different aspects based on the stakeholder's concern. Consider again the change shown in Figure 1. Application developers who cannot access `setName()` would deem this change to be a refactoring since the existence of `setName()` is removed from their concerns. On the other hand, almost all library developers who can freely access `setName()` would not accept this change as a refactoring. Consequently, the observable behavior of the changed program differs although the same change was made to the program. In this case, a programmer who made the change wants to discriminate between application developers' concerns and library developers' ones.

To accommodate individual concerns that emerge in refactoring, we introduce the concept of a frame that represents the boundary of each stakeholder's concern about the refactored code. A frame helps a programmer discriminate code changes that intertwine with various concerns. She must observe the changed code inside a particular frame but can ignore that outside it. In this context, *frame-based refactoring* preserves the observable behavior within a frame.

Figure 3 depicts the concept of frame-based refactoring. S_1 and S_2 denote snapshots of the code before and after a change, respectively. F_A is a frame which user U_A is conscious of. To be precise, such a logical frame with respect to the scope of program elements is called a *spatial frame*. $\langle S, F \rangle$ represents the observable behavior of partial code (specified program elements) existing in frame F of snapshot S. $\langle S, * \rangle$ represents the observable behavior of the whole code (all the program elements) of S. In a traditional manner, this change is not a refactoring for both U_A and U_B. On the other hand, the change is a refactoring for U_A in the frame-based manner (and is still non-refactoring for U_B).

In the current status of our work, each frame is simply defined as a set of test cases. Test cases are in general linked to use cases, which are created based on stakeholder's actions. In other words, a frame is responsible for testing uses cases for a particular stakeholder. In Figure 3, for example, U_A is an application developer who usually accesses limited modules (classes or packages) through their respective interfaces. In this example, the library urges U_A to use the interface that presents just `register()` and `getName()` shown in Figure 1. This indicates that U_A does not know the existence of `setName()`, and wants the test cases T1 and T2 shown in Figure 2 to pass. In this case, F_A can be defined as a set of T1 and T2. On the other hand, if U_B is a library developer who can access all the methods within the library, he is always concerned with T3 and T4 in addition to T1 and T2. In this case, different frame F_B (not appearing in Figure 3) is prepared for U_B, which is

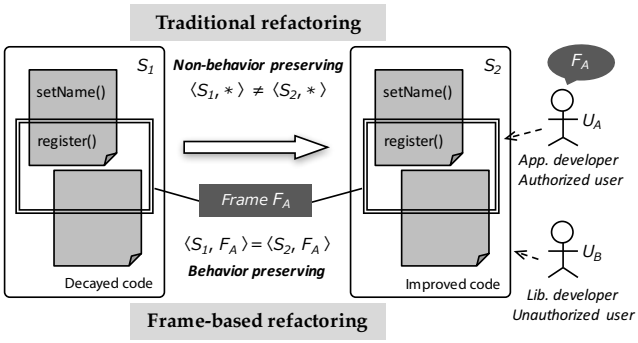

Fig. 3. Traditional refactoring versus frame-based refactoring.

defined as a set of these four test cases. As another example, different frames can be created for an authorized user and an unauthorized user (attacker) if individual test cases with respect to the users are well-defined or the prepared test cases are properly divided into ones for each of the users.

Using frames, a programmer can leave a more useful commit message that explicitly describes information about the frame when she commits the changed code. For example, the detailed description of "*refactoring with respect to frame F defined as test cases T1 and T2*" is a better message than simply described "*refactoring*" and "*refactoring with a behavioral change*". A more detailed message helps her coworkers understand the change.

To make it easier for programmers to form frames they need, we are tackling the implementation of two mechanisms. One expects programmers to assign specific annotations labeled with frame names to their corresponding test cases, and the other adopts the on-demand collections of test cases depending on the visibility of methods to be tested. A refactoring tool manages (e.g., displays, retrieves, and joins) frames and guarantees that all results of test cases within the same frame are preserved during refactoring. Unfortunately, we do not have a solid evidence for the benefits from frame-based refactoring yet. Moreover, we are exploring a variety of concerns that form the frames and finding a new way to define them.

Acknowledgment

This work was sponsored by the Grant-in-Aid for Scientific Research (15H02685).

References

[1] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[2] M. Hafiz and J. Overbey. Refactoring myths. *IEEE Software*, 32(6):39–43, 2015.

[3] M. Kim, T. Zimmermann, and N. Nagappan. An empirical study of refactoring challenges and benefits at Microsoft. *IEEE TSE*, 40(7), July 2014.

[4] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE TSE*, 30(2):126–139, 2004.

[5] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *Proc. ICSE '09*, pages 287–297, 2009.

[6] W. F. Opdyke. Refactoring object-oriented frameworks. Technical report, Ph.D. thesis, University of Illinois, Urbana-Champaign, 1992.

[7] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *Proc. ICSE '12*, pages 233–243, 2012.