An Approach to Analyzing Execution Preservation in Java Program Refactoring

Thi-Huong Dao[1], Hong Anh Le[2(✉)], and Ninh Thuan Truong[1]

[1] VNU, University of Engineering and Technology, Hanoi, Vietnam
{huongdt.di12,thuantn}@vnu.edu.vn
[2] Hanoi University of Mining and Geology, Hanoi, Vietnam
lehonganh@humg.edu.vn

Abstract. Code refactoring is a technique that improves the existing code in order to make software easier to understand and more extensible without changing the external behavior. Software design patterns, programming language independent reusable solutions to comment problems, are well-known in Java communities. On one hand, the refactoring using design patterns brings many benefits such as cost saving, flexibilities, and maintainability. On the other hand, it potentially causes bugs or changes execution behavior of Java programs. This paper proposes a new approach to checking behavior preservation properties of Java programs after applying design patterns. We present new definitions to compute pre/post conditions of program behavior. In the next step, the paper makes use of Java Modeling Language (JML) to represent and check if the refactored program neglects to preserve the external behavior. A motivating example of Adaptive Road Traffic Control (ARTC) is given to illustrate the approach in detail.

Keywords: Refactoring · Design patterns · Consistency · ARTC

1 Introduction

Software development is an elaborated and time-considerable process involving many steps in which maintenance phase plays an important role. This phase takes a high cost if the designs or codes are poor that makes software difficult to understand. As a consequence, developers find hard to maintain, modify, or create new features.

Software refactoring, originally introduced by Opdyke in his dissertation [10], is techniques, which are widely adopted for improving existing designs or codes without altering the external behavior. It includes a series of small transformations restructuring the software system. The software is expected to execute correctly as same as it does before refactoring.

During refactoring process, if developers realize smell codes, they will find solutions to improve these with a new structure. Design patterns [6] are general repeatable reusable solution to a commonly occurring problem within a given

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2017
P. Cong Vinh et al. (Eds.): ICCASA 2016, LNICST 193, pp. 101–110, 2017.
DOI: 10.1007/978-3-319-56357-2_11

context in software design. It is a description or template for how to solve a problem that can be used in many different situations.

Refactoring to Patterns is the process of improving the design of existing code with patterns, the classic solutions to recurring design problems. Refactoring to Patterns suggests that using patterns to improve an existing design is better than using patterns early in a new design. We should improve designs with patterns by applying sequences of low-level design transformations, known as refactorings. Patterns are language independent, they have been broadly used in many programming languages including Java.

However, a big problem with design patterns in refactoring process is that we can not assure the execution behavior of the original program and its refactored one is consistent. It means that some execution may not be preserved in new one. Several approaches have been proposed to checking the consistency between programs using graph transformation techniques [2,12], and XML metadata interchange [5]. In this paper, we propose an approach to checking the consistency of execution behavior of Java programs before/after refactoring. The main contributions of the paper are (1) defines how to compute pre/post conditions of software execution behavior (i.e. program scenario), (2) utilize JML notations and tools to describe the constraint in refactored program to check the consistency property, and (3) illustrate the proposed approach with a case study of ARTC program.

The rest of the paper is organized as follows. In the following section, we review related work. Section 3 gives an overview of Strategy pattern and JML. A motivation example of Adaptive Road Traffic Control system is displayed in Sect. 4. Section 5 presents the approach to checking consistency software refactored programs. Section 6 concludes and gives some directions for future works.

2 Related Work

As mentioned earlier, William Opdyke [10], whose first introduced the *refactoring* term for object-oriented software, opened a new research direction in the field of software engineering. His research interested in describing the prerequisites and automatic program restructurings required to guarantee preservation of behavior. Moreover, he also developed a refactoring tool for Smalltalk.

According to analyzing of different criteria (e.g. the activities of refactoring, the specific techniques and formalisms), Tom Mens et el. [9] provided an extensive overview of existing research area. Particularly, they also discussed about various formal techniques which are used in refactoring process, such as invariants, pre/postcondition, graph transformation, program slicing, software metric, etc. Their research was very valuable with others people whose study in refactoring.

In [4], JML was used to specify the behaviors of a Java program. With the support tools [3] the program will be static and dynamic testing. Static testing will return syntax error and the invalided type of variables, dynamic testing (run-time assertion checking) will notify all kinds of run-time assertion violations.

Nevertheless, these studies were only done purely on a software program and have no related to refactoring process.

Some studies represent the behaviors of the software system via the assertions (invariants, pre/postconditions) [10,11], one problem is that the static checking of some preconditions may require very expensive analysis, or may even be impossible.

In comparison to prior works, our approach focuses on analyzing execution preservation between original program and evolution itself in refactoring process. Based on the JML specification to represent pre/post-conditions of a scenario, we use OpenJML tool to automated checking these constraints on both programs and answer the question whether the refactored program is satisfied initial behaviors specification or not? The advantages of our work is feasibility in experiment and semi- automated in checking behavior consistency.

3 Background

We interested in design pattern term as well as Java Modeling Language (JML) as a theory foundation to execute our method in the next section.

3.1 Strategy Pattern

Design pattern, is a general reusable solution to a commonly occurring problem within a given context in software design, has become popular since 1994 by GOF [6], in which they have categorized the design patterns into three groups, namely creational, structural, and behavioral patterns.

In this section, we clarify one behavioral pattern, namely **Strategy pattern**. It *encapsulates, defines a family of algorithms and makes them interchangeable independently from clients*. The strategy object changes the *executing algorithm* in the particular context object.

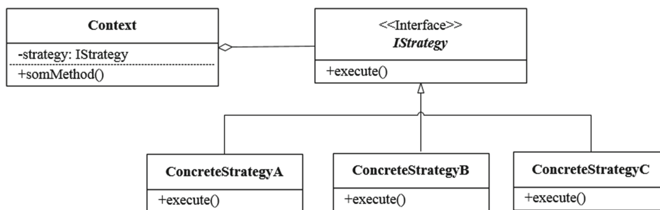

Fig. 1. Strategy pattern

The **Strategy pattern** has three participants as shown in Fig. 1:

- **IStrategy:** The interface that is shared among the concrete strategy classes in the family. Class *Context* uses this interface to call the algorithm defined by a concrete strategy.

– **ConcreteStrategy:** Where the real implementation of strategy takes place.
– **Context:** The class maintains a reference of type *IStrategy*. In some cases, *Context* may implement operations so that *ConcreteStrategy* can access its data.

The advantage of using strategy pattern is that encapsulating algorithms in individual classes will render reusing code much more convenient and hence, the behavior of the **Context** can be altered at run-time dynamically.

3.2 Java Modeling Language

Java Modeling Language (JML) [7] is a *behavioral interface specification language* (BISL) that can be used to specify Java classes and interfaces. JML specifications or assertions can be added directly to source code as a special kind of comments called annotation comments, or they can live in separate specification files. These assertions are usually written in a form that can be compiled, so that their violations can be detected at run-time.

The two main advantages in using JML are [7]:

– the precise, unambiguous description of the behavior of Java program modules (i.e., classes and interfaces), and documentation of Java code,
– the possibility of tool support [3].

JML's syntax is very close to the Java programming language, so it easily used by programmers who have familiar with Java. In this section, we only present a brief overview about functions as well as features of JML. For more details, one can refer to [8].

4 A Motivating Example: Adaptive Road Traffic Control System

4.1 ARTC System's Description

Traffic congestion is an ever increasing problem in towns and cities all over the world. Local authorities must continually work to maximize the efficiency of their road networks and to minimize any disruptions caused by accidents and events.

From the object-oriented perspective, the initial ARTC system is described by a simplistic model with four classes, namely *Detector, TrafficController, Road* and *Optimizer*. The UML class diagram of initial ARTC system is shown in Fig. 2.

The UML sequence diagram have accomplished the task of showing how the objects interact with each other in scenario. We will portray our approach with the two significant methods: *gettrafficFlow()* and *optimizeTraffic()*. The sequence diagram for Scenario of functional processes is depicted in Fig. 3.

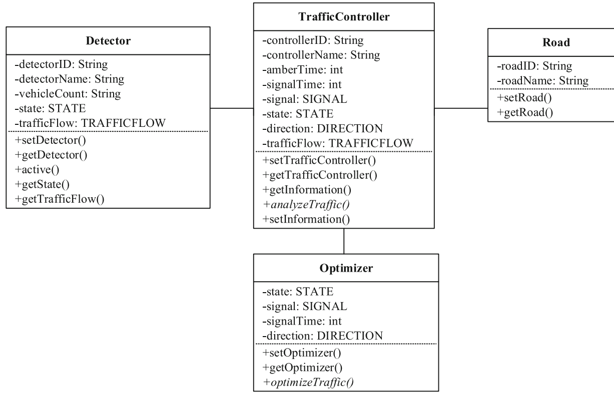

Fig. 2. The initial class diagram of ARTC system

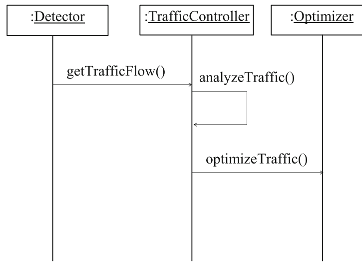

Fig. 3. Sequence diagram for calculating optimal control

4.2 Selected Patterns

In Fig. 2, the method *optimizeTraffic()* belong to class *Optimizer*, which is employed to optimize light signals of the ARTC system. However, the system design may have following problems with the *optimizeTraffic()* of the class *Optimizer*:

- Algorithms are so **complex** to implement in one, therefore make the source code as large and arduous to maintain.
- It takes **time** as well as **effort** to add new algorithms to the existing ones.
- The code of the existing algorithms are **difficult to reuse**, especially when one want to create a hierarchy from *Optimizer* class.

In order to overcome these limitations and improve the system, we are going to optimize it by using **Strategy pattern**. As illustrated in Fig. 4, we detach three optimization strategies (*SignalOptimizeStrategy, TimeLimitOptimizeStrategy, AdjacentOptimizeStrategy*) from the class *Optimizer* then formed a hierarchy of algorithm classes that share the interface *OptimizerStrategy*. After applying Strategy pattern, the sequence diagram of the scenario calculating optimal control is re-drawed in Fig. 5.

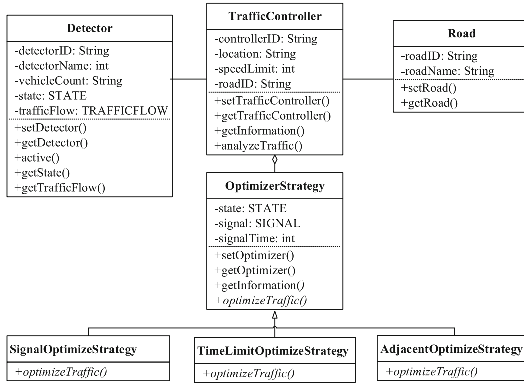

Fig. 4. Class diagram of ARTC system after applying Strategy pattern

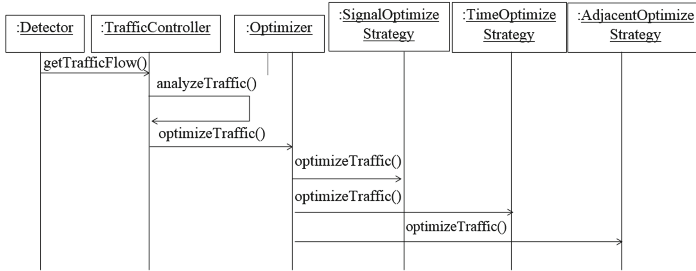

Fig. 5. Sequence diagram for calculating optimal control after applying Strategy pattern

4.3 Behaviour Preservation

ARTC system has a real time characteristic because of immediate responses to variant of traffic flow conditions, some identified constraints need to be preserved are[1]:

- If the **state** is *heavyTraffic* and the **signal** is *red*, it will be ensured that the **signal** is turned to *green*.
- If the **state** is *lowTraffic* and **signal** is *red*, it will be ensured that the **signal-Time** is increased.
- If the **state** is *highTraffic* and **direction** is *noChoose*, it will be ensured that the **direction** is turned to *choose*.

If we implement these constraints as a purely Java code, it may be not enough to guarantee the correct behavioral execution. As a consequence, we employ the JML' code to annotate it in order to two purposes, firstly, ensuring the performance of the source code is correctly, secondly, automated validation of software programs.

[1] Due to space considerations, we do not display completed constraints here.

5 Approach to Checking Consistency

5.1 Formal Representation of a Software Program

In order to automated checking the consistency between two programs after applying design patterns in refactoring, we introduce a formal representation of a software program as follows:

Definition 1 (Program). *A program P is formally represented by a 2-tuple $\langle C_P, S_P \rangle$, where C_P is a set of classes and S_P is a sequence of method invocation statements in the main body of a program.*

Definition 2 (Class). *A class $C_{i_P} \in C_P$ is represented by a 3-tuple $C_{i_P} = \left\langle M_{C_{i_P}}, A_{C_{i_P}}, I_{C_{i_P}} \right\rangle$, where $M_{C_{i_P}}$ is a set of public methods, $A_{C_{i_P}}$ is a set of public attributes, and $I_{C_{i_P}}$ states a set of class invariants.*

Definition 3 (Method precondition). *The precondition $PRE_{m_{i_P}}$ of the method $m_{e_i} \in M_{C_{i_P}}$ in the class C_{i_P}, is a condition that it has to satisfy when it begins to execute.*

Definition 4 (Method postcondition). *The postcondition $POST_{m_{i_P}}$ of the method $m_{e_i} \in M_{C_{i_P}}$ in the class C_{i_P}, is a condition that it has to satisfy after executing.*

In Definition 1, S_P is a set of sequence statements of invoking methods which begins with the entry point of the main function in program P. In this paper, we employ the "*scenario*" term to refer the S_P signal.

Definition 5 (Scenario). *A scenario S_P is represented by a 4-tuple $S_P = \langle C_{S_P}, PRE_{S_P}, M_{S_P}, POST_{S_P} \rangle$, where $C_{S_P} \subseteq C_P$ represents a set of classes involved in the scenario, PRE_{S_P} is the scenario precondition, M_{S_P} is a sequence of methods of involved classes, and $POST_{S_P}$ states the scenario postcondition.*

Definition 6 (Scenario method). *A method in the scenario is a 4-tuple $M_{kSP} = \left\langle PRE_{k_{S_P}}, M_{k_{S_P}}, POST_{k_{S_P}}, k \right\rangle$, where $PRE_{k_{S_P}}$ states the method precondition, $M_{k_{S_P}}$ is the public method of the involved in the scenario, $POST_{k_{S_P}}$ is the method postcondition, and k is the execution order of method in the scenario.*

In this paper, we consider the case that pre/postcondition of a method is the conjunction of predicates on the attributes of classes involved in the scenario, i.e., $PRE_{k_{S_P}} = \bigwedge P(A_{CijP})$, where $A_{CijP} \in A_{CiP}$ is a attribute, $CiP \subseteq C_{SiP}$ and P is predicate. A scenario consists of a sequence of methods, hence its pre/postcondition are formed by their pre/postcondition. A scenario pre/postcondition is defined on pre/postconditions of all methods involved in the scenario as follows.

Definition 7 (Scenario precondition). *The scenario precondition PRE_{S_P} is defined by the precondition of the first happened method in the scenario.*

The precondition of the first method in the scenario specifies constraints of all scenario-related public attributes.

Definition 8 (Scenario postcondition). *The scenario postcondition $POST_{S_P}$ is defined by the conjunction of the constraint on public attribute A_{C_P} in the method postcondition $POST_{k_{S_P}}$ of the last happened method in M_{S_P}.*

Let a scenario $S = (m_1, m_2, ..m_n)$, where m_i, $i = \overline{1..n}$, is the i-th method happened in the scenario. From Definition 6, we have $m_i = (pre_{mi}, m_{mi}, post_{mi}, i)$ and $post_(mi) = \bigwedge P_k(A_k C)$, where P_k are the predicate on $A_k C$, which is the attribute of class C involved in the scenario. Assume that the scenario has one public attribute A_C that appears in both postconditions of two methods m_i and m_j such that $1 \leq i < j \leq n$. Then we have $post_{mi} = P_i(A_C)$ and $post_{mj} = P_j(A_C)$. Since m_i happens before m_j, $P_j(A_C)$ must be hold after executing the j-th method.

Definition 9 (Refactor). *A refactor R using design patterns is denoted $R : P \overset{D}{\longmapsto} P'$, where P and P' are the original program and its evolution, respectively, D is the applied pattern name.*

5.2 Consistency Rules of Program Refactoring

In Subsect. 5.1, we address that the pre/postcondition of a scenario can be computed from pre/postcondition of involved operations. In practice, the execution of a scenario must be preserved its pre/postcondition.

Proposition 1 (Execution preservation of original program). A program P is said to be execution preservation if with each scenario, its preconditions and its postconditions are preserved before and after execution, respectively.

Formally, $PRE_{S_P}[S_P]POST_{S_P}$.

Proposition 2 (Execution preservation of refactored program). A refactored program P' is said to be execution preservation with the original one P if with the same scenario execution, its preconditions are preserved before and its postconditions are hold after execution.

Formally, $PRE_{S_P}[S_{P'}]POST_{S_P}$.

In this proposition, the scenario pre/postcondition of the refactored program are figured out through the scenario one of the original program according to Definitions 7 and 8.

5.3 Applying the Proposed Approach to Check the Execution Preservation of ARTC System

Back to the example in Sect. 4, all initial behaviors specification of the ARTC system have been validated checking on Eclipse software by plug-in OpenJML.

Now, after refactoring, we are going to consider whether evolution program is satisfied all behaviors specification of initial program or not?

In experiments, we have carried out the implementation the source code of the ARTC system after refactoring. Based on the set of rules which was built in these Section, we have shown the pre/postcondition of the evolution scenario as well as checked the constraints on it. The experimental results are illustrated in Fig. 6.

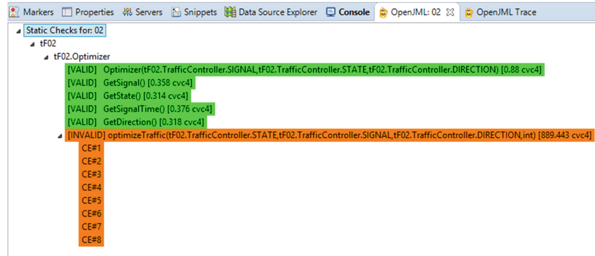

Fig. 6. The result of checking behavior preservation on refactored program

In other words, the refactored program not preserves all behaviors of initial program in execution, so it should consideration how to the corrected refactoring process.

6 Conclusion and Future Work

In this paper, we have proposed an approach to verify the execution preservation of refactored program which is performed by design patterns in software program. In addition, JML is used to describe constraints of class behaviors. We have proposed consistent rules to verify if the scenario execution of the original program and refactored one is preserved the same constraints in evolution process.

It has been many works to check the consistency of a program, however, these works focus on the consistency between different phases of life cycle development model (e.g., implementation and design phase) or different diagrams of a model (e.g., state diagrams and sequence diagrams), but our research pays attention in checking consistency between original program and its evolution in the implement phase.

To demonstrate the approach, we have implemented a program of an ARTC system in UML. In the case study, we have just illustrated only the consistency verification when applying Strategy pattern in the only a pair of scenario, respectively in the both programs, others scenarios may be done in a similar way for the more complex system.

As portrayed in the case study, we can see that the calculation of pre and post-condition of scenarios is time-consuming and error-prone if we do it manually. For the future works, we will adopt tools to calculate automatically constraints and verify the program evolution process.

References

1. Alexander, C., Ishikawa, S., Silverstein, M.: Pattern Languages. Center for Environmental Structure, vol. 2 (1977)
2. Bottoni, P., Parisi-Presicce, F., Taentzer, G.: Coordinated distributed diagram transformation for software evolution. Electron. Notes Theoret. Comput. Sci. **72**(4), 59–70 (2003)
3. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Rustan, K., Leino, M., Poll, E.: An overview of JML tools and applications. Int. J. Softw. Tools Technol. Transfer **7**(3), 212–232 (2005)
4. Cok, D.R.: OpenJML: software verification for Java 7 using JML, OpenJDK, and Eclipse. arXiv preprint arXiv:1404.6608 (2014)
5. Dong, J., Sheng, Y., Zhang, K.: A model transformation approach for design pattern evolutions. In: 2006 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, ECBS 2006, pp. 10–92 (2006)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
7. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. ACM SIGSOFT Softw. Eng. Notes **31**(3), 1–38 (2006)
8. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., et al.: JML reference manual (2008)
9. Mens, T., Tourwe, T.: A survey of software refactoring. IEEE Trans. Software Eng. **30**(2), 126–139 (2004)
10. Opdyke, W.F.: Refactoring: a program restructuring aid in designing object-oriented application frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign (1992)
11. Roberts, D.B., Johnson, R.: Practical analysis for refactoring. University of Illinois at Urbana-Champaign (1999)
12. Zhao, C., Kong, J., Zhang, K.: Design pattern evolution and verification using graph transformation. In: 2007 40th Annual Hawaii International Conference on System Sciences, HICSS 2007, p. 290a, January 2007