A Test Model for Design Pattern Application

Nien Lin Hsueh *
Department of Information Engineering and Computer Science, Feng Chia University
Taichung, Taiwan
nlhsueh@mail.fcu.edu.tw

Abstract

In the current trend, design pattern has been widely used for improving software quality. However, using design patterns is not easy, developers need to understand their complex structure and behavior, and have to apply them in the correctly. Therefore, several approaches are proposed to check the violence of pattern application in a system. We argue only static checking to patter structure is not enough, dynamic testing to test the patterns' semantics is necessary. In this paper, we propose a test model for design patterns. We explore the potential error point of each design pattern thoroughly, and provide a testing guideline for each pattern application.

1 Introduction

In the current trend, design pattern has been widely used for improving software quality. In the academic research, design patterns is one of important research area and applied in different areas [7, 16, 14], many design pattern detection approaches are proposed [8, 17, 19], some work discuss pattern quality metrics [10, 9, 2], design pattern formalization and specification [23, 12], and its benefits [13, 11, 21, 20].

However, using design patterns is not easy, developers need to understand their complex structure and behavior, and have to apply them in the right way. Based on my teaching experience and some research reports, we know some patterns are applied in the wrong way [18, 15]. Therefore, several approaches are proposed to check the violence of pattern application in a system [3, 22]. We think only static checking to pattern structure is not enough, dynamic testing to test the patterns' semantics is necessary.

In this paper, we propose a test model for design patterns. We explore the potential error point of each design pattern thoroughly, and provide a testing guideline for each pattern application.

For example, when we apply *Singleton* pattern, we should conduct the following tests: creating two objects and check if they have the same references. Another example is *Strategy*, when we applied strategy a with this paper, and replace it by another algorithm b, they should have the same behavior, that is, the same output. The following code shows the testing context:

```
1 class TestStrategyDesignPattern {
```

*This research was supported by the National Science Council, Taiwan R.O.C., under grants MOST 108-2221-E-035-028.

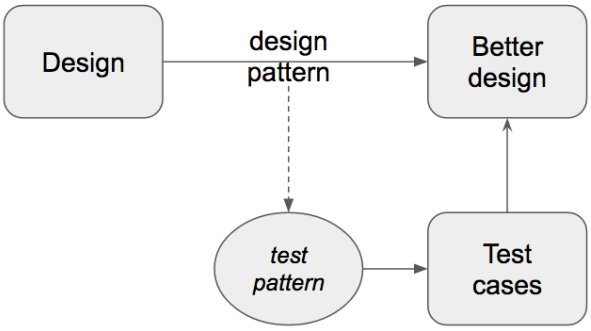

圖 1: The concept of *test pattern*

```
2     void testSameResult() {
3           Context c = new Context();
4           Strategy a = new StrategyA();
5           c.setStrategy(a);
6           int resultA = c.doIt();
7           Strategy b = new StrategyB();
8           c.setStrategy(b);
9           int restultB = c.doIt();
10          assertEquals(a, b); // should have same result
11    }
12 }
```

We use JUnit-like approach to test the Strategy-applied system. Different design pattern has different properties, therefore the test methods are also different. In this paper we will explore possible approach for them.

1.1 Paper outline

The remainder of this paper is structured as follows: In section 2, we describe the related work to this research. Section 3 introduces the details of our approach by two design pattern examples. Section 4 summarizes our approach and future work.

2 Related work

2.1 Pattern testing

Chu and Hsueh proposed test refactoring approach in a pattern driven developent [5]. Test-first strategy and code refactoring are the important practices of Extreme Programming for rapid development and quality support. The test-first strategy emphasizes that test cases are designed before system implementation to

keep the correctness of artifacts during software development; whereas refactoring is the removal of "bad smell" code for improving quality without changing its semantics. However, the test-first strategy may conflict with code refactoring in the sense that the original test cases may be broken or inefficient for testing programs, which are revised by code refactoring. In general, the developers revise the test cases manually since it is not complicated. However, when the developers perform a pattern-based refactoring to improve the quality, the effort of revising the test cases is much more than that in simple code refactoring (see Fig. 2). To address this problem, the composition relationship and the mapping rules between code refactoring and test case refactoring are identified, which infer a test case revision guideline in pattern-based refactoring. A four-phase approach to guide the construction of the test case refactoring for design patterns is developed.

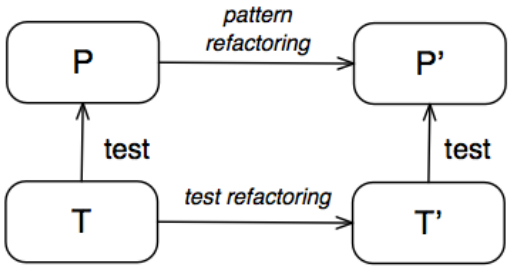

圖 2: Pattern based test case refactoring

To understand the application context for a pattern in a system, Lin proposed a *Design Pattern Unit Test (DPUT)* approach, which utilizes java Annotation skill to record the pattern utilization and verifies with the expection in the DPUT. This research also design a software framework to help developers design the DPUP in a specification basis. The code is implemented as an Eclipsed plug-in which can automatically transform DPUT into class diagram for better understanding. For the system maintainers they can find out the errors in the earlier phases.

2.2 Pattern evaluation

Huston [10] provides an analysis method to examine whether design patterns are compatible with design quality metrics. Each pattern solution has a non-pattern solution that provides a simple solution if that pattern is not adopted. The comparison of the metric score returned for a non-pattern solution against that for a pattern solution is employed for examining the compatibility. For example, the *Mediator* design pattern is compatible with the coupling factor metric (COF) if the pattern is intended to promote loose coupling, and the COF degree of a design with *Mediator* is significantly less than that of design without the pattern. The research results display little pattern-metric conflicts, that is, using design patterns can reduce high metric scores, which might otherwise cause low-quality alarm.

Hsueh etc. al [9] improves Huston's approach and propose a general approach to verify if a design pattern is well-design. The approach is based on the object oriented quality model. They decompose a design pattern into functional requirement and non-functional requirement parts, both of them have related structure to realize the requirements. A *quality focus* is also defined to formally identify the intent of the

design pattern. If the quality focus is not consistent with the structures, the design pattern will be seen as a conflicting design.

In other efforts, there have been attempts to quantify differences between using patterns and non-pattern versions in different contexts, such as game development [1] or a software engineering course[4]. Ampatzoglou and Chatzigeogiou use a qualitative and a quantitative approach to evaluate the benefits of using patterns in game development [1]. They perform the evaluation on two real open-source games under the versions of implementation with pattern and without pattern. The results of experiments show that the application of patterns can reduce complexity and coupling, as well as increase the cohesion of the software. Chatzigeorgiou et al. [4] assign each student team attended the software engineering course to deliver a software application with and without patterns for assessing students' comprehension and the benefits of patterns . In addition to collecting problems from students' reports, they also ask students to provide the measured values with 7 metrics for comparing the non-pattern and the pattern versions. The experiment reveals some points related to the efficient learning of design patterns in a software engineering course.

3 Our approach

To design the test model for pattern testing, we have to consider the following issues:

- Correctness and Precision. Correctness considers if all faults can be explored by the proposed test cases, and Precision considers if the test can explore the faults in an efficient way. Our test model should satisfy correctness and precision.
- Coverage. In general we have many coverage strategies for testing, for example statement coverage or branch coverage. When we test we should enlarge the coverage as we can. Therefore we should care about the normal cases, boundary cases and exception cases. When we utilize this concept to testing a pattern-applied system, what are the boundary cases and exception cases?

Potential Pattern Violation (PPV) To test the pattern-applied application more efficiently, we propose the concept of *Potential Pattern Violation (PPV)*. Each pattern has different potential violation that may happen in the application.

Test Pattern for Design Pattern (TP4DP) One of the benefits of design pattern is it is well structured and contains executable code for programmers. Our TP4DP follows the same concept. A test pattern has the following sections:

- Pattern name: the pattern under test;
- Potential pattern violation: the possible issues when applying the pattern;
- Testing guideline: the guideline to test the pattern;
- Demo code: same code to test the pattern.

In the following, we describe the concept by examples.

3.1 Testing "*Singleton*" pattern

Pattern Name: Singleton The intent of the Singleton is to ensure a class has only one instance, and provide a global point of access to it.

Potential Pattern Violation There are two PPV in the Singleton pattern:

- PPV1: When we create two objects from the same class, they refer to different objects.
- PPV2: The original object constructor is not set to be private.

Testing guideline

- To test PPV1: Creating two objects of the same Singleton class, check if there are equal by using the operator "==".
- To test PPV2: Using code review to test this violation. Observer if the original object constructor is set to private, if No, the design does not pass the PPV2 test. Another approach is using static check by Java reflection mechanism.

Demo code (PPV1) Here is the demo code for the class *Radio*:

```
1 class TestRadio {
2     void testRadioSingletonPPE1() {
3         Radio r1 = Radio.instance();
4         Radio r2 = Radio.instance();
5         assertTrue(r1==r2): "Singleton is violated";
6     }
7 }
```

3.2 Testing "*Observer*" pattern

Pattern name: Observer The intent of *Observer* is to "define a one-to-many dependency between objects so that when one object changes state, all its dependent are notified and updated automatically." To achieve this aim, the pattern encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.

The Observer object should register to the subject initially. When the subject changes state, the registers (observers) will get notified and react to the message. Note that all observers implement the same interface (called Observer) such that the Subject can communicate with them.

Potential Pattern Violation When we implement this pattern, we may make error when we have single or many observers registering to the subject, especially on the extreme case- the first and the last observer and when no observer registers to the subject.

- PPV1: When *Subject* changes state, the *Observer* object does not get notified. For the boundary testing, we do two more tests:
  - PPV1.1: The first register does not get notified;
  - PPV1.2: The last register does not get notified;
- PPV2: No objects register to the subject. The system behaves abnormally when the subject object changes state.
- PPV3: The Subject is coupled not only to the Observer base class but also its subclasses.

Guideline To test PPV1, we should create some *Observers* and register to the subject, and then change state to cause the event trigger. Check if the observer update its view or state. In most cases the observers update its state by changing GUI view and is difficult to check by program, in this case we can check by our direct observation (by eyes). Note that we should be careful to see the first and last observer to meet the PPV1.1 and PPV1.2.

To test PPV2, we change the state of the *Subject* before the *Observers* register to the *Subject*, and then check if any abnormality occur. PPV3 needs a static verification to check if the *Subject* class navigate the concrete observer class. The design pattern checking tool (such as *pattern4*) can identify what is the *Observer* class, and then we verify if the *Subject* navigates (refers to) any *Observer* implementation classes.

Demo code To test PPV1 and PPV2, we create some *Observers* and check if the first and last observer will get the notification.

```
1  class TestObserver {
2      void testObserverPPV1() {
3          Subject s = new Subject();
4          Observer obs1 = new XObserver();
5          Observer obs2 = new YObserver();
6          Observer obs3 = new YObserver();
7          s.addObserver(obs1);
8          s.addObserver(obs2);
9          s.addObserver(obs3);
10         s.changeState();
11     }
12     void testObserverPPV2() {
13         Subject s = new Subject();
14         s.changeState();
15     }
16 }
```

4 Conclusion

In this paper we propose an idea of pattern test for guiding the programs with patterns applied. We believe a program is error-prone when they are designed in delicate and complex structure. Design patterns are

good for flexible design but not easy to understand and apply. For pattern beginners we need a guideline, framework or tool to help them. We demonstrate two examples of *Singleton* and *Observer* to illustrate our ideas. In the future we will explore more design patterns and develop a framework for the pattern test.

References

[1] Apostolos Ampatzoglou and Alexander Chatzigeorgiou. Evaluation of object-oriented design patterns in game development. *Information and Software Technology*, 49(5):445–454, 2007.

[2] Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Sofia Charalampidou, and Paris Avgeriou. The effect of gof design patterns on stability: a case study. *IEEE Transactions on Software Engineering*, 41(8):781–802, 2015.

[3] Alex Blewitt, Alan Bundy, and Ian Stark. Automatic verification of java design patterns. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 324–327. IEEE, 2001.

[4] Alexander Chatzigeorgiou, Nikolaos Tsantalis, and Ignatios Deligiannis. An empirical study on students' ability to comprehend design patterns. *Computers & Education*, 51(3):1007–1016, 2008.

[5] Peng-Hua Chu, Nien-Lin Hsueh, Hong-Hsiang Chen, and Chien-Hung Liu. A test case refactoring approach for pattern-based software development. *Software Quality Journal*, 20(1):43–75, 2012.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[7] Alan R Graves and Chris Czarnecki. Design patterns for behavior-based robotics. *IEEE Transactions on Systems, Man, and cybernetics-part A: systems and humans*, 30(1):36–41, 2000.

[8] Dirk Heuzeroth, Thomas Holl, Gustav Hogstrom, and Welf Lowe. Automatic design pattern detection. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 94–103. IEEE, 2003.

[9] Nien-Lin Hsueh, Peng-Hua Chu, and William Chu. A quantitative approach for evaluating the quality of design patterns. *Journal of Systems and Software*, 81(8):1430–1439, 2008.

[10] Brian Huston. The effects of design pattern application on metric scores. *Journal of Systems and Software*, 58(3):261–269, 2001.

[11] Clemente Izurieta and James M Bieman. A multiple case study of design pattern decay, grime, and rot in evolving software systems. *Software Quality Journal*, 21(2):289–323, 2013.

[12] Salman Khwaja and Mohammad Alshayeb. Survey on software design-pattern specification languages. *ACM Computing Surveys (CSUR)*, 49(1):21, 2016.

[13] TH Ng, SC Cheung, WK Chan, and Yuen-Tak Yu. Do maintainers utilize deployed design patterns effectively? In *Proceedings of the 29th international conference on Software Engineering*, pages 168–177. IEEE Computer Society, 2007.

[14] Michael Oduor, Tuomas Alahäivälä, and Harri Oinas-Kukkonen. Persuasive software design patterns for social influence. *Personal and ubiquitous computing*, 18(7):1689–1704, 2014.

[15] Lutz Prechelt, Barbara Unger-Lamprecht, Michael Philippsen, and Walter F Tichy. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *Software Engineering, IEEE Transactions on*, 28(6):595–606, 2002.

[16] Cagri Sahin, Furkan Cayci, Irene Lizeth Manotas Gutiérrez, James Clause, Fouad Kiamilev, Lori Pollock, and Kristina Winbladh. Initial explorations on design pattern energy usage. In *Green and Sustainable Software (GREENS), 2012 First International Workshop on*, pages 55–61. IEEE, 2012.

[17] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T Halkidis. Design pattern detection using similarity scoring. *Software Engineering, IEEE Transactions on*, 32(11):896–909, 2006.

[18] Peter Wendorff. Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In *Software Maintenance and Reengineering, 2001. Fifth European Conference on*, pages 77–84. IEEE, 2001.

[19] Marco Zanoni, Francesca Arcelli Fontana, and Fabio Stella. On applying machine learning techniques for design pattern detection. *Journal of Systems and Software*, 103:102–117, 2015.

[20] Cheng Zhang and David Budgen. What do we know about the effectiveness of software design patterns? *Software Engineering, IEEE Transactions on*, 38(5):1213–1231, 2012.

[21] Cheng Zhang and David Budgen. A survey of experienced user perceptions about software design patterns. *Information and Software Technology*, 55(5):822–835, 2013.

[22] Chunying Zhao, Jun Kong, and Kang Zhang. Design pattern evolution and verification using graph transformation. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 290a–290a. IEEE, 2007.

[23] Hong Zhu. On the theoretical foundation of meta-modelling in graphically extended bnf and first order logic. In *Theoretical Aspects of Software Engineering (TASE), 2010 4th IEEE International Symposium on*, pages 95–104. IEEE, 2010.