FIT REFACTORING-IMPROVING THE QUALITY OF FIT ACCEPTANCE TEST

Xu Liu

A Thesis

Submitted to the graduate college of Bowling Green
State University in partial fulfillment of
The requirement for the degree of

Master of Science

August 2007

Committee:

Joseph Chao, Advisor

Ron Lancaster

Mohammad Dadfar

ABSTRACT

Joseph Chao, Advisor

Acceptance tests are formal testing conducted to determine whether a system satisfies its acceptance criteria or not and whether the acquirer should accept the system or not. A suite of acceptance tests for large projects might include a large number of test cases; therefore, automation of acceptance test is in great demand. Framework for Integrated Tests (FIT) is a popular tool employed in Agile Software Development to automate acceptance tests. Its most attractive feature is that it uses customer readable tables as test cases so that customers can write test cases. Refactoring is the process of restructuring or rewriting code without changing its interface and functionality. Refactoring make the code easier to read, understand and maintain, and sometime helps to improve the performance of the system. In a typical project that uses FIT as an acceptance test tool, the size of FIT acceptance tests grows as the size of system code grows, and the acceptance design may go far away from the original design (this may happen in any project, not restricted in a project using FIT). At this stage, it would be difficult to read and maintain the FIT acceptance test, and it is time to improve the quality of the acceptance test.

In this research, we introduce the concept and reveal the importance of FIT Refactoring. Several FIT Refactoring methods are introduced as examples to show the needs of FIT Refactoring and the methods how it can be accomplished. Of course, the methods given here are some obvious refactoring methods, and new methods can be discovered in further research. We

also discuss the proper time to do FIT refactoring and proper efforts that should be devoted in it. The similarities and differences between system code refactoring and FIT acceptance test refactoring are also one part of the research. During the research, there are some unexpected findings. One of them is that sometimes, the bad code in FIT acceptance test indicates bad code in the system code.

ACKNOWLEDGMENTS

I would like to give some sincere thanks to my adviser, Dr. Joseph Chao. Without his guidance and support, it would be impossible for me to complete this research. His experience in research, willingness to help and patience are great helps to me. I also want to thank my committee members, Dr. Ron Lancaster and Dr. Mohammad Dadfar for reading and providing valuable suggestions to this research.

TABLE OF CONTENTS

Page

TABLE OF CONTENTS v

LIST OF FIGURES vii

LIST OF TABLES viii

LIST OF TABLES viii

CHAPTER 1 INTRODUCTION 1

1.1 Agile Software Development 1

1.2 Acceptance tests 2

1.3 Framework for integrated tests (FIT) 3

1.4 Refactoring 8

CHAPTER 2 LITERATURE REVIEW 11

2.1 Acceptance Test and FIT 11

2.2 Refactoring on System Code and Test Code 16

CHAPTER 3 FIT REFACTORING 19

3.1 Why FIT Refactoring 19

3.2 Table Refactoring and Fixture Refactoring 21

3.3 When to do FIT Refactoring 23

CHAPTER 4 METHODS OF FIT REFACTORING 26

4.1 FIT refactoring methods on tables 27

4.1.1 Rename 27

4.1.2 Introducing new domain concept 30

4.1.3 Extract similar part of tables into fixture 33

4.1.4 Merge tables 36

4.1.5 Split table 38

4.2 FIT refactoring methods on fixtures 40

4.2.1 Merge similar methods 40

4.2.2 Extract method 43

4.2.3 Extract similar methods in fixture into new class 45

CHAPTER 5 OTHER FINDINGS 49

5.1 Improper FIT acceptance test may indicate improper system code 49

5.2 Recommendation on organization of FIT acceptance test table 51

CHAPTER 6 CONCLUSION 57

REFERENCES 59

LIST OF FIGURES

Figure Page

Figure 1-1 FIT acceptance Test Case Sample 5

Figure 1-2 FIT acceptance Test Fixture Sample 6

Figure 1-3 FIT acceptance Test Result Sample 8

Figure 4-1 Part of AddInventory Fixture 28

Figure 4-2 AddInventory Fixture Before Refactoring 29

Figure 4-3 AddInventory Fixture After Refactoring 29

Figure 4-4 Refactored purchaseBeverageNew Table 36

Figure 4-5 EditRecipe Fixture Before Refactoring 42

Figure 4-6 EditRecipe Fixture Before Refactoring 42

Figure 4-7 PurchaseBeverage Before Refactoring 45

Figure 4-8 PurchaseBeverage After Refactoring 45

Figure 4-9 PurchaseBeverage Before Refactoring 46

Figure 4-10 setDefaultRecipes Class 47

Figure 4-11 purchaseBeverageNew Fixture 47

Figure 4-12 PurchaseBeverage Test Case 48

Figure 5-1 coffeeMaker Class Diagram 50

Figure 5-2 Part of setUpCoffeeMaker Fixture 50

LIST OF TABLES

Table Page

Table 4-1 Part of AddInventory Test Case 28

Table 4-2 AddInventory Table Before Refactoring 29

Table 4-3 AddInventory Table After Refactoring 29

Table 4-4 offteredServices Table 31

Table 4-5 lineCapability Table 31

Table 4-6 The Refactored offeredServices Table 32

Table 4-7 New offteredServices Table 32

Table 4-8 New offeredServices Table After Refactoring 33

Table 4-9 purchaseBeverageNew table 35

Table 4-10 CalculateCredit table 37

Table 4-11 CalculateVIP Table 37

Table 4-12 CalculateCredit Table 38

Table 4-13 The New CalculateCredit Table 39

Table 4-14 Refactored New CalculateCredit Table 40

Table 4-15 Refactored New CalculateVIP Table 40

Table 4-16 Part of EditRecipe Before Refactoring 41

Table 4-17 Part of EditRecipe After Refactoring 41

Table 5-1 coffeemaker.AddRecipe Table Before Refactoring 53

Table 5-2 New AddRecipe Part 1 55

Table 5-3 New AddRecipe Part 2 55

Table 5-4 New AddRecipe Part 3 55

Table 5-5 New AddRecipe Part 4 56

CHAPTER 1 INTRODUCTION

FIT Refactoring is based on FIT (Framework for Integrated Tests) and refactoring. FIT is a tool to automate acceptance test and sometimes is used to document business requirement. Created by Ward Cunningham [19], FIT is getting more and more popular in Agile Software Development (being introduced in the following section). Refactoring is typically done on system code to maintain good structure of the code, but the acceptance test can benefit from it as well. In this paper, FIT refactoring will be introduced, and it helps to keep a good design of your tests and requirements.

1.1 Agile Software Development

Agile Software Development is a framework, or a set of methods for developing software systems. Different from the traditional, inflexible and plan-driven waterfall model, Agile Software Development emphasizes on frequent communication with customers and flexibility throughout the life cycle of development. Agile Software Development was first promoted by a non-profit organization, Agile Alliance, which was founded in late 1990s [22]. The idea of Agile was the combination of several methodologies, such as Adaptive Software Development (ASD), Extreme Programming (XP) [1], Scrum [18] and Crystal Clear. Although the traditional waterfall model is still widely used in the software industry now, more and more people are attracted by Agile Software Development.

The characterized activities in Agile Software Development include simple design, pair programming, adapting user stories, small releases, short iterations, test-driven, customer provided acceptance test, no documentation, and most of them seems contradictory to the traditional models. Although those activities are not only used in Agile Software Development, but also other models including waterfall, they are highly recommended in many Agile methodologies. A lot of these activities are widely used, not only by Agile Software Development project teams, but also by many non-Agile teams.

1.2 Acceptance tests

Acceptance tests, also called integrated tests, are formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the acquirer to determine whether or not to accept the system. It is a high level test on business operations. A suite of acceptance tests contains many test cases, usually thousands of them.

Compared with unit test, there are several significant differences between these two important forms of tests. First, programmers use unit tests to test the internal or technical elements of the code [15]. If the code is not working properly, finding out the defects is done with the help of unit test. On the other hand, acceptance test tries to demonstrate that, with certain input, the system will produce correct outputs, and these correct outputs are expected by testers and customers. If there are problems, often, the defects are not inside a class or method; they are usually between classes or methods or lie in the misunderstanding of business logical, because the defects inside a class should have been corrected by unit test already. Second, unit

test is usually performed on a certain class or method. While in acceptance test, developers test the entire system or a logically related block of code. Finally, as Agile Software Development requested, acceptance test should be done by customers or customers and testers together, but unit test is used for programmers to test their own code. Often, in acceptance test, testers translate the user provided test cases into program readable format, and then, the test cases are executed.

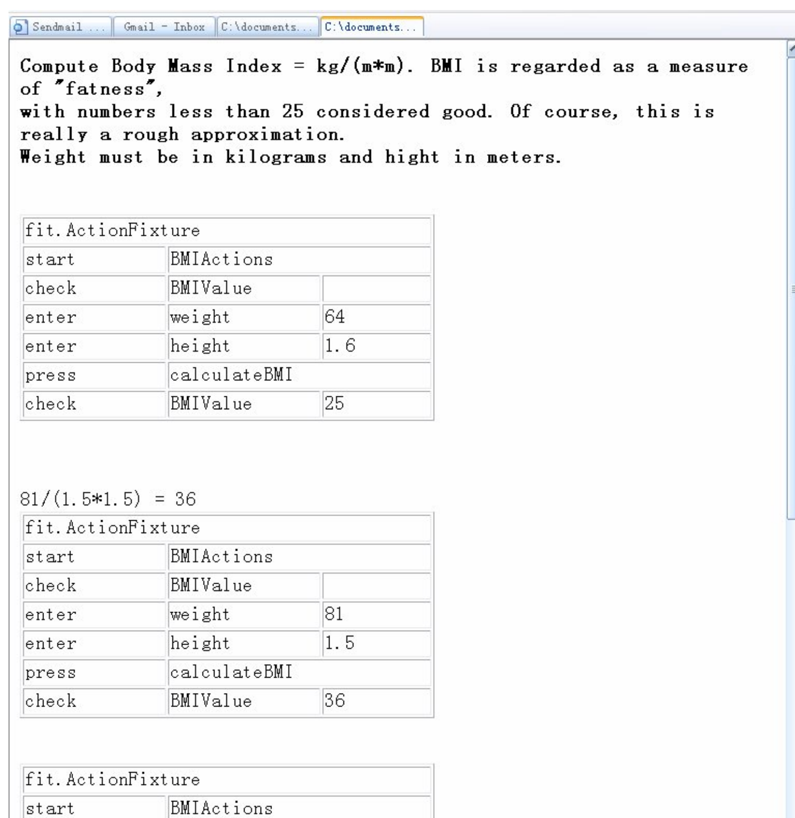
Acceptance test is important for customers in that it helps them to decide whether the software is acceptable; it is also important for developers because that it helps regressively test the code for quality control [5]. One major benefit of acceptance test is that it helps the collaboration between customers and developers, which is emphasized by XP [1] and other Agile methods. Customers need to provide valuable test data for acceptance test so that the test can "talk" to the system under test. And testers need to act as "translators" to make customer written test cases understood by the system. When the system "replies", customers and testers need to make sure the "answers" of the system are what they expect, so that customers can accept the system with confidence.

1.3 Framework for integrated tests (FIT)

Performed on the whole system and containing thousands of test cases, acceptance test for a large project is usually a time consuming task. If acceptance test is executed manually, we most likely can not afford the regressive execution of the huge number of test cases. Tools have been

developed to automate acceptance test. FIT framework is one of the most popular tools in the Agile Software Development world.

FIT acceptance test cases consist of two parts. One part of it is human-readable tables, in which customers can put test case data in those tables. The tables can be in the form of tables in a HTML file, a Microsoft Excel file, a Microsoft word document, in Microsoft Windows Notepad, or even in a Wiki (a popular web collaboration tool where everyone can edit the content of the page [4]). Figure 1-1 shows some samples of FIT input tables (test cases). Using table as the input form and communication tool is one of the benefits of FIT framework. Different from programming languages such as Java and C, customers without any knowledge in programming can understand the information presented by tables in applications such Microsoft Word. Besides, those applications mentioned above are easily accessible understandable for them. This reduces the difficulty of communication between customers and developers, and good communication is of great value for a successful project.



Compute Body Mass Index = kg/(m*m). BMI is regarded as a measure of "fatness",
with numbers less than 25 considered good. Of course, this is really a rough approximation.
Weight must be in kilograms and hight in meters.

fit.ActionFixture		
start	BMIActions	
check	BMIValue	
enter	weight	64
enter	height	1.6
press	calculateBMI	
check	BMIValue	25

81/(1.5*1.5) = 36

fit.ActionFixture		
start	BMIActions	
check	BMIValue	
enter	weight	81
enter	height	1.5
press	calculateBMI	
check	BMIValue	36

fit.ActionFixture		
start	BMIActions	

Figure 1-1 FIT acceptance Test Case Sample

The other part of FIT is a set of test fixture(s), which is usually implemented by developers since it requires certain knowledge of the system implementation. They contain the logic of how the test data will be passed into the system under test and how the software is executed. Fixtures are codes that fetch the data in the test table specification and feed the system under test with the data. There are several types of fixtures such as RowFixture, ColumnFixture, ActionFixture, RowEntryFixture, and CommandLineFixture. Figure 1-2 is an ActionFixture that handles the test case shown in Figure 1-1. Each type of test fixtures has its own usage and pattern [19]. ColumnFixture, ActionFixture and RowFixture are the most frequently used fixtures.

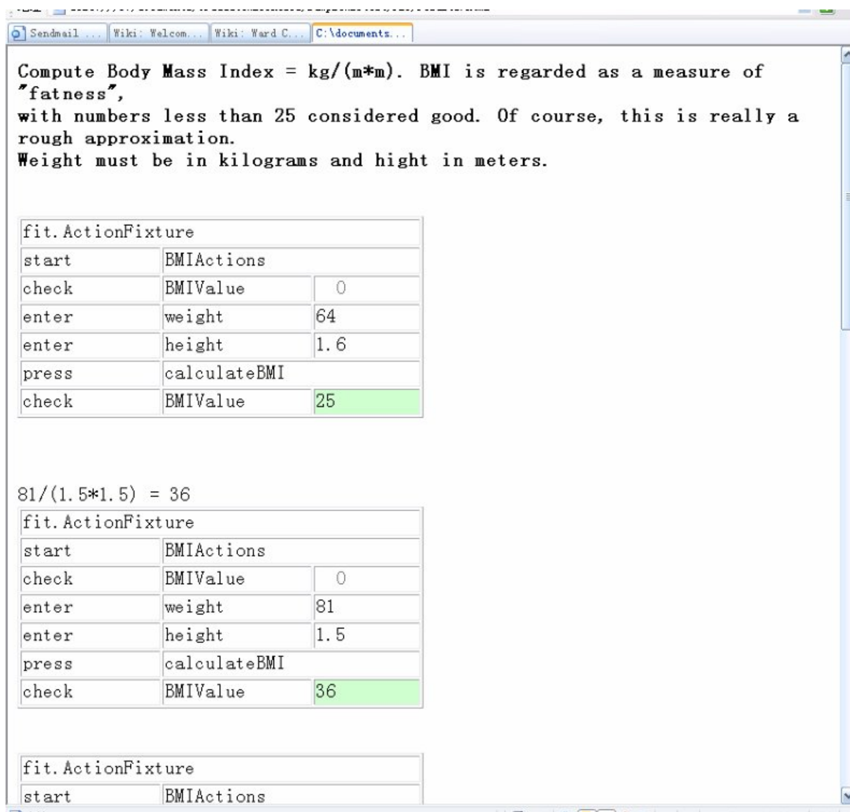
```
import fit.ActionFixture;
import fit.ScientificDouble;
import java.lang.Math;

public class BMIActions extends fit.Fixture{
    private BMIPanel BMISample = new BMIPanel();
    private double Currentweight = 0.0;
    private double Currentheight = 0.0;
    private double BMIValue = 0.0;

    public int BMIValue(){
        return (int)Math.round(BMIValue);
    }
    public void weight(double weight){
        this.Currentweight = weight;
    }
    public void height(double height){
        this.Currentheight = height;
    }
    public void calculateBMI(){
        this.BMIValue = BMISample.computeBMI(Currentweight,
Currentheight);
    }
}
```

Figure 1-2 FIT acceptance Test Fixture Sample

- ColumnFixture: deals with calculations. It is designed to test whether the calculation result is as expected. A good example is a program that calculates the total price of item while we know the item prices and quantities. The fixture takes in the two parameters (count and unit price), and calls the system under test to calculate the result, and then compares the system result with the expected result.
- ActionFixture: deals with business processes. Designed to check whether a sequence of actions made to a system have the desired effects to the system under test. There are some reserved words for ActionFixtures including start, enter, press and check. In test specification tables, these reserved words are used to perform operations on the system under test. One example may be a login dialog box. The test case is to enter user name and password, press login button, and then check the login succeeds.

- RowFixture: deals with testing list-like result. When you need to test a program with the results of a search or a query of the system under test, you need to use RowFixture. For example, the test case may check whether the persons in a chatting room are the ones that are expected.

FIT is particularly useful at testing from a business perspective [14]. After executing the test cases against the system under test, FIT provides a result that looks similar to the input tables. The test results for the test cases in Figure 1-1 are shown in Figure 1-3. The main difference between the input and output tables is that the output table will show the execution result and displaying the comparison of the inputted. In a result table, it is easy to distinguish whether the result is what you are expecting since they are color coded. If the result is exactly what you have expected, then the cell containing the result is highlighted in green; otherwise, the cell is in red containing both the expected answer and actual answer to alert you. If there are exceptions when the system under test is running, that cell will be in yellow and the description of exception will be listed. If the cell is in gray, it means the data is ignored for some reasons. The colors in the result tables are helpful and very easy to understand by customers, and thus enable the effective collaboration between developers and customers.



Compute Body Mass Index = kg/(m*m). BMI is regarded as a measure of "fatness",
with numbers less than 25 considered good. Of course, this is really a rough approximation.
Weight must be in kilograms and hight in meters.

fit.ActionFixture		
start	BMIActions	
check	BMIValue	0
enter	weight	64
enter	height	1.6
press	calculateBMI	
check	BMIValue	25

81/(1.5*1.5) = 36

fit.ActionFixture		
start	BMIActions	
check	BMIValue	0
enter	weight	81
enter	height	1.5
press	calculateBMI	
check	BMIValue	36

fit.ActionFixture	
start	BMIActions

Figure 1-3 FIT acceptance Test Result Sample

1.4 Refactoring

Refactoring is a technique based on object-oriented programming. It changes the structure of existing code in small steps, meanwhile, the interfaces and functionalities remain the same [14, 16]. Not aiming at improving the system performance [17] (but sometimes refactoring can improve the system performance), refactoring makes the existing code more understandable, easier to read, to maintain and to add new functionalities to meet new requirements [20]. Typically, refactoring is performed on system code.

A good structure for a software system is important, but as new requirements come to the developers, as modification is done on the code, the code becomes more complex and more

different from what it was originally designed. So the structure of the system is deteriorates. Refactoring helps to maintain and improve good structure of the code.

Actually, most programmers perform refactoring daily. When you finish a class, or a block of code, you come back to review the code. You find some of the code is not well structured and make an effort to clean it up. For example, a method in the class has a name making no sense to you, so you decide to give it a new name, which should not only make sense to you but also to others. While changing names (renaming) might be the simplest refactoring, it is one of the most important and frequently performed refactoring method.

Usually, the piece of bad code you want to refactor on is called a “smell” by people in the “Agile” world. In a kitchen, if a fish smells, then it means the fish is bad and most likely, not eatable. Same thing happens to software system code. If the code “smells”, it would be wise to restructure the code.

There are many refactoring methods and renaming, as mentioned above is one of the simplest methods. For example, move a method or a field from one class to another more relative class, extract a block of code into a new class to make the code better structured, store logically related data in a data class in it to clean up a mess of data, are all useful refactoring methods. Often, those methods are used together to make the structure of system design clear. There are quite a few refactoring tools available in the field. Eclipse the popular Java IDE, has built-in refactoring tools, Refactoring Browser [23] is a tool that can be used for IBM Smalltalk, ReSharper [24] is a refactoring tool based on .NET platform, and nearly all popular object-oriented programming languages have some refactoring tools available.

When refactoring, there are things that need to be aware of. First, do not change the behavior of a class. If the behavior of the class is changed, then it is not refactoring, though it may also help. Second, make sure the code performs correctly before changing anything, and it is important to have test suit to test the code. Third, do refactoring in small steps. After each small step, execute the test suit again to make sure the code still works, and then continue the refactoring.

Different from the system code refactoring mentioned above, in this research, we are introducing refactoring on FIT acceptance test, and we call it FIT refactoring. Since FIT, contains tables and fixtures as test cases, refactoring can be applied to both tables and fixture code. The benefits of refactoring on system code also work on FIT acceptance test-it keeps the test case clear, understandable and well structured. It is important to keep test case well structured; especially FIT test cases are often used not only for acceptance testing but also for business requirement documentation.

CHAPTER 2 LITERATURE REVIEW

FIT was first created in 1997 and has been gaming to popularity in recent years especially in Agile world. It is relatively new compared with other acceptance testing tools, so researches on FIT have been abundant and researches on FIT Refactoring are non-existence. In this chapter, we present literatures on FIT, on acceptance test and on refactoring in the following sections.

2.1 Acceptance Test and FIT

Although considered important in all software system projects, acceptance tests are often neglected in many projects. Since the test code is not shipped to customers as part of the final product, there is a strong temptation to give up acceptance test, reduce the number of test cases dramatically and do the acceptance tests manually and sporadically when the time frame for the project is tight. In [13], Meszaros et al proposed "Test Automation Manifesto" where automated tests should be concise, self checking, repeatable, robust, sufficient, necessary, clear, efficient, specific, independent, maintainable and traceable. This manifesto provided good directions for refactoring acceptance test cases.

It has been an important point emphasized by Agile Software Development that a successful project requires effective communication between customers, developers and testers [10]. Many researchers have shown that FIT Framework could help with the communication well with a language (tables and additional specifications assisting the tables) that all of the three roles in the project can understand, instead of a programming language like Java or C, which might not be

understood by customers. FIT provides a way to make customers and developers understand the system much more precisely [19].

Melnik, Read and Maurer [11] conducted an experiment with college students at two universities: University of Calgary and Southern Alberta Institute of Technology (SAIT). They listed some common challenges for functional requirements specifications, because of which, software projects often fail. The common challenges of requirement specifications included in the experiments were:

- Noise (information not relevant to the problem)
- Silence (important aspects are not mentioned)
- Over-specification (aspects of the solution are mentioned as part of the problem description)
- Wishful thinking (a problem difficult or impossible to address)
- Ambiguity (not clear)
- Reader subjectivity
- Forward references (mention aspects of a problem not yet mentioned)
- Oversized documents (difficult to understand, use and maintain)
- Multiple representations
- Little to no user involvement.

Through their experiments, they found that most of the common challenges of functional requirement documents were solved by using FIT, but FIT also made few of them even worse. The unaddressed problem was silence, where using FIT made the problem even worse according

to the experiments. There were also other problems such as that some of the test tables and fixtures of the user stories implemented by students were “fat” which means too many logic in the fixtures. However this could be explained by their inexperience of using FIT Framework.

In their successor research [15], the authors were interested in finding out how to use FIT in both quality assurance and to represent functional requirements in a test-first environment, which means writing test cases before production code. The same as their previous research, the experiments were also conducted at the same two universities. In the experiments, FIT was used as the main requirement specification tool. In one course, FIT was introduced at the beginning of the semester, but in the other, it was introduced half way through the semester. Students were required to implement functions to pass the tests, and then extend these tests with additional scenarios. After the experiment, the authors provided four categories of usage patterns on FIT. First, strategies of test fixture design. It included how subjects construct FIT tables and fixtures, what kind of fixtures to use, RowFixture or ActionFixture; and also a discussion of fatness when using FIT. The second was the strategy of using test-suits against single tests, which examines the organization of FIT tests. Test-suits were proper to be used as regression tests and it could test whether one function works well with others, while single tests gave better response time. Third, development approaches, identified subject actions during development. Most subjects used FIT tests to drive their projects. Finally, Robustness of test specification, how subjects dealt with exceptional cases was analyzed. Typos or omissions were easily detected by subjects and did not seem to affect the delivery of projects according to the result of the experiment.

Read, Melnik and Maurer [16] continued their experiments at the same two universities. This time they used FIT as both acceptance test tool and requirement specifications, and students were also required to write additional test suits. The authors concluded that FIT tests can be used to replace requirement specification documents. Survey showed that most students answered that FIT acceptance tests was a good method of defining functional requirements, and was easier than defining a formal requirement specification. Students also found that FIT tests were somewhat easier or at least no more difficult than writing JUnit tests.

Throughout the series of their experiments, one of the consensus they formed was that the learning curve of using FIT Framework was not prohibitively steep (people won't use a tool that is too hard to learn). Most of the students learned to use the basic fixtures (RowFixture, ColumnFixture and ActionFixture) in one or two weeks. The other common conclusion is that FIT documents (test cases) working as requirement specifications were adequate. The requirement specifications described in FIT tests cases written by the instructors could be easily understood by students and implemented by students with little background in FIT.

Gandhi et al [10] shared their experience of using FIT as the only set of requirement specification documents in their project. Their conclusions included:

- Because the team was not familiar with FIT at the beginning, they encountered some difficulties in collaboration between team members and stakeholders.
- They were distracted by the syntax of FIT documents and lost the focus on developing the most appropriate specification.

- They had is that business analysts should not use plain test as their product, but produce FIT documents; story specification should be written prior to the story implementation.
- The success of the FIT acceptance tests make the developers more confident on refactoring code.
- Throughout the use of FIT documents, the authors found that FIT documents led them to discover new domain concepts.
- It was good to break complex business rules into smaller, simpler ones.
- FIT documents (with additional descriptive test if need) should be the main definition of the specification. It should be kept clean and free from duplication.
- When there is a bug or a change of the specification, update the current FIT documents instead of creating a new one.
- After integration, tests need to be run correctly again. Organizing the specification is difficult.

The authors tried several ways to categorise the test cases in different directories; they changed their methods in different development iterations. How to name the tests (use story numbers or named intelligibly) also needs discussion. The authors introduced subroutines (called in main FIT documents) in FIT test case to reduce the duplication, and this worked pretty well. It also improved the readability. The authors tried to go further on using subroutines, but they found that using default parameters and nested subroutines made it overly generic and difficult for analysts to read. Actually, some of the conclusions they made were basic FIT Refactoring methods.

When new developers joined the team, the finished FIT documents helped them to catch up with the development and the coverage of the FIT documents made them aware of any unanticipated work. Test-Driven Development (TDD) method might make developers write code simply to make the test pass, but without closely collaborating with the original user stories. It would be helpful if the same developer who prepared the FIT documents (user stories) work on the same user stories. If there were not enough business analysts, they would not feed the developers (developers may need more user stories to implement), and then the analysts would have to work more hours on analyzing business requirements rather than communicate with developers, which led to less communication between them. So the team should be balanced.

2.2 Refactoring on System Code and Test Code

The first research on refactoring is considered by William Opdyke in his 1992 PhD dissertation [8]. He defined refactoring as "a set of program restructuring operations that support the design, evolution and the reuse of object-oriented application frameworks." He also introduced a set of refactoring methods, including both low-level refactorings and complicated high-level refactorings. Based on object-oriented programming, he provided refactoring operations in the following 3 categories:

- Refactoring to generalize: creating an abstract superclass.
- Refactoring to specialize: subclassing, simplifying conditions.
- Refactoring to capture aggregations and components.

In 1999, the book *Refactoring: Improving the Design of Existing Code* [7] by Martin Fowler and coauthors Kent Beck, John Brant, William Opdyke and Don Roberts was published. In this book, the authors explained when to use refactoring, how to use refactoring and the rules that need to be obeyed when refactoring.

Refactoring methods are explained in a systematic manner in this book, they are sorted in the following catalogs:

- Composing methods
- Moving features between objects
- Organizing data
- Simplifying conditional expressions
- Making method calls simpler
- Dealing with generalization
- Big refactorings

Refactoring on system code has been discussed frequently by researchers while refactoring on test code is a relatively new. Since testing was given more and more attention, refactoring on test code and test cases came to the view of software industry in recent years. In Agile Software Development projects, testing is constantly emphasized, and because of the flexibility of Agile development, test cases are often changed frequently. To maintain a good structure, we need refactoring on test cases.

Refactoring on unit test is discussed in [3], [12]. In [3], the authors found that test code usually has a distinct set of bad code, based on how the test cases are organized, implemented

and interacted with each other. Because of the distinction of styles of bad code, the corresponding refactoring methods are different from that for system code. In their definition of test refactoring, they emphasized that test cases should not be added or removed, and refactoring should make them more understandable, readable and/or maintainable. They also pointed out that one should not underestimate the importance of having fresh test code, which is very important for new programmers on a team to understand the system as well as the test cases. So it is necessary to put enough efforts on test cases refactoring. The authors also provided a list of bad smells in unit test and the refactoring methods on those smells.

CHAPTER 3 FIT REFACTORING

We define FIT Refactoring as the process of applying refactoring techniques to the FIT acceptance tests, without adding or removing test cases. The purpose is to maintain a good structure of acceptance test cases and make them understandable, readable and maintainable. In this chapter, we will discuss the needs and the benefits of FIT Refactoring; introduce the two categories of FIT Refactoring (refactoring on tables and refactoring on fixtures), and also describe the differences and similarities between system code refactoring and FIT Refactoring.

3.1 Why FIT Refactoring

There are several reasons for refactoring on FIT acceptance test cases. In a software project, acceptance test is very important, and it needs to be well organized which means understandable readable and maintainable. There are reasons the design of the test cases are changed. In a typical project that uses FIT as acceptance testing tool, the number of test cases may increase dramatically as the system code grows, and this growth can break the original design. When this happens, it could be difficult to maintain the test cases, and new functions to the system may make the acceptance design change. One feature of Agile Software Development is that the blueprint at the very beginning is simple and may not be a good one. The acceptance test design may not be a good design also, so the change of its structure in the project is necessary. Readable and understandable FIT acceptance test is very helpful for a programmer just joining a team. It can help the newcomer to catch up easily and understand the project rapidly, so we need the FIT

acceptance test design to be good. One method to make the design good is refactoring which is mentioned by Martin Fowler in [7], doing refactoring can help one to understand the project better. What's more, in some projects, FIT is used as business requirement documentation and this documentation should be easy to read. So refactoring on FIT acceptance test can help to make test cases easier to read, understand and maintain, and a good acceptance test should be exactly like that. For of all these reasons, it is necessary and beneficial to do FIT Refactoring on FIT acceptance test.

Acceptance test is crucial to a software project on two aspects. One is testing aspect and the other is on the documentation aspect. From testing aspect, it decides whether the customer will accept the work produced by developers or not. In most cases, although all the functions are tested by the unit test, when come together in integration, there may still be problems among those classes and functions. Acceptance test can help to make sure all parts of the project and those conjunctions between parts work correctly. In Agile projects, acceptance tests are executed regressively in iterations. Those repeated acceptance tests make sure that the new functions are working properly and so does the updated previous iteration code.

From the documentation aspect, the FIT acceptance test is also important. In a project using FIT acceptance test driven methodology and using FIT as the business requirement documentation, a clear, understandable, well structured test is even more important than in other projects. When using acceptance test driven development, developers write code according to the test case tables, in which the class names, major function names and main frames are already defined. So it is obvious that the test has a significant influence on the system code. Also, a

successful run of the test against the system code greatly increase the confidence of developers. If there are newcomers joining the project group, FIT acceptance test would be a great tool to help them catch up with the others in the development as confirmed in [10]. So the FIT acceptance is important and worth some efforts to do it well.

As refactoring on system code improves the structure of the code, FIT Refactoring provide the same benefit works to FIT acceptance test cases. Throughout the software development cycle, acceptance test cases evolve constantly. The customer may add new requirements, new functions, or new test cases from time to time, so the design will most likely keep changing and drifting away from the original. While the structure of the acceptance test and the structure of the system keep changing, it is likely that the system grow into a complicated mess. Refactoring will maintain the good design structure and prevent this from happening. FIT refactoring, that makes the FIT test more readable has never been discussed systematically before.

3.2 Table Refactoring and Fixture Refactoring

Since FIT acceptance test cases contains two parts: tables and fixtures, there are two types of refactoring: table refactoring and fixture refactoring. However, these two types of refactoring can not be separated clearly simply because tables and fixtures are two inseparable parts of FIT acceptance test. A change in the table may require the corresponding change in the fixtures and vice versa. For the refactoring methods on fixtures, they are often similar to the refactorings on system code. If acceptance test is executed manually or other acceptance testing tools, there will

be no tables involved in test cases, or those tables are read by humans but not machines. So table refactoring is unique in FIT Refactoring.

There are similarities and differences between system code refactoring and fixture refactoring in FIT. Most of the popular refactoring methods in [7] work well on fixtures, for example “Rename”, “Remove Assignments to Parameters” and “Reduce Scope of Variable”. The differences lie in the special characters of test fixtures. Test fixtures do not usually have complex class hierarchies; therefore the refactoring methods dealing with those class hierarchies, such as “Collapse Hierarchy”, “Extract Superclass”, “Encapsulate Collection”, and “Change Bidirectional Association to Unidirectional” are not often used here. Also, FIT fixtures should not be “fat”, so there is seldom complex logic in them. Logical statements or conditions are usually in system code. So the refactoring methods in [7] improving logic may not be applicable in FIT Refactoring. Examples are “Decompose Conditional”, “Remove Double Negative” and “Replace Conditional with Polymorphism”.

Since refactoring of FIT tables is unique and there are no tables in system code, there is no comparison between system code refactoring and table refactoring. Most of the unique methods in FIT Refactoring are in table refactorings. Table refactorings deal with how we structure the acceptance test cases, how to organize the data we used and how to call the methods in the system under test. In a way, FIT table structuring is similar to how tables are organized in a relational database system. Table refactoring works similarly to database normalization.

3.3 When to do FIT Refactoring

When is a good time to do refactoring? For refactoring on system code, as Martin Fowler stated in his book [7 pp. 58-60], we usually do not spend a large amount of time just to refactoring. Refactoring is something you do when you see the need for it. For example, when adding features to the system, which might be a function, a new class, or some new business rules, or when trying to fix a bug or receiving a code review, you may want to refactor the system code when you see the need to do it.

Similar to system code refactoring, it would not be a good practice to set up one or two weeks time specially for doing refactoring. It should be done when there is bad code in your acceptance test cases. If the acceptance test is working well at the moment, then there is no need to do refactoring. One possible time might be when you are reviewing the FIT acceptance test tables and fixtures, and this is possible because you might be using it as business requirement documentation for an ongoing project. To make sure the system is built according to the requirement of the customers, you may look at the requirements frequently. Any time you find something that is considered as bad code, and it is doing damage to the test cases, you do refactoring on it. Martin Fowler mentioned a ‘three times’ practice. The first time you discover some code that is not good and you are not sure whether should do refactoring or not, leave it; the second time you find it, leave it there again; but if you find the same piece of bad code the third time, do refactoring on it. Actually, if this piece of code is obviously doing a lot of harm to your acceptance test cases, refactoring the code should be done immediately; otherwise, for other

situations, the ‘three times’ practice could be a good idea. Another possible time you may want to do refactoring on acceptance test cases is when you have just run the test cases against the system under test. That is the time when it is likely to find something that is going wrong, so you want to fix the bug, or when there is a strong feeling that the test case can be structured better.

For FIT refactoring, the time to do it also depends on which part of the test cases you want to refactor. The time for table refactoring and fixture refactoring could be different. But there is one more thing to notice: making sure the code is working properly before you do refactoring is one of the most important principles of refactoring [7]. For FIT tables created by customers, if the customer believes they are correct and ready to go, it is time for table refactoring. For FIT fixtures, which is written by developers, we need to discuss a good time when we can do refactoring on them.

When you want to do refactoring on the FIT acceptance test fixtures, it is necessary to make sure the test fixtures are working right before refactoring. If you change the structure of the code before you know that the code is working properly, that would be more of a code design rather than refactoring (though both of them help) so the question is the same as when can we make sure those fixtures are working properly. When and how can we make sure the acceptance tests are good to run? The simplest answer is that we have the system code available, and then we run the acceptance test against it. If the results of the FIT acceptance test are what we expect, then the FIT acceptance test and system under test are working properly.

But is it necessary to wait until the system under test is ready, and only after that we can do refactoring on the acceptance test cases? Of course not. Acceptance test is built and run

continuously in each iteration of a project, especially in an Agile Software Development project. During the iteration, when the domain code is not finished, you can run part of the acceptance test or you can run it with the help of mock objects [2] or stub objects [6]. After executing the acceptance test cases for the first time and getting satisfying results, it means that the acceptance test is ready for refactoring if there is a need. There is no need to wait until the end of the iteration when the code under test is fully completed.

In test driven development, we have the acceptance test before we have domain code. In order to make sure the acceptance test code is working properly and ready for refactoring, it might be a good idea to run those tests before the domain code is ready? What mock object or stub can help us to manage this, there are some drawbacks. For example, using many fake objects would be hard to manage, and mock objects could make test more fragile. Also, building large amount of mock objects or stubs takes a lot of time, and it could be a waste even though it may offer good acceptance tests as a result.

As a conclusion, the best time for FIT refactoring could be after we have run it against the domain code in the iteration. FIT Refactoring too early in an iteration may have little effect on the domain code, and mock objects and stubs could be too time-consuming to be worthwhile.

CHAPTER 4 METHODS OF FIT REFACTORING

FIT, as a good tool to automate acceptance test, has two parts: tables and fixtures. These two parts together make FIT acceptance test possible. We can do refactoring on both of them, and the methods are different. For refactoring on fixtures, it is similar to system code refactoring, but there are some differences. Usually system code is complex, and there are hierarchies, but FIT fixtures are relatively simple, and there are fewer sup classes or sub classes. So fixture refactoring should be easier than system code refactoring. On the other hand, acceptance test table refactoring is different from refactoring on code. This restructure happens not only on code, but mostly on tables. It is a new type of refactoring compared to fixture refactoring, so refactoring on test case tables is emphasized in this research.

In this section, we introduce several methods to demonstrate the concept of FIT refactoring. For each of the methods, we provide some examples. Many of these examples are based on the sample FIT test cases for the CoffeeMaker project created in an online FIT tutorial [21]. Since we will be referring to the CoffeeMaker project throughout this Chapter, a brief introduction of the project is given below.

The coffeeMaker is a system simulating a coffee maker machine. It allows administrators to add coffee, milk, and sugar into the machine, and add, edit or delete recipes for beverages such as coffee, hot chocolate, and latte. A recipe includes the name, price and how much of each

ingredient are in the beverage. Then, after the system is ready to serve, customers can purchase the beverage using the machine.

4.1 FIT refactoring methods on tables

FIT table refactoring is a new domain on refactoring. In the following methods we developed, some of them are related to the name of the table, some are related to table structures, and some are related to data organization in the table.

4.1.1 Rename

Naming style is an important part of software development. The system code is not only read by machines, but more importantly it is read by humans. Therefore, mnemonic (or meaningful) names for classes and methods are essential to the software development team. A reasonable and meaningful name provides a visual cue and makes the code more readable.

FIT can also benefit from having good names for all relevant classes, methods or variables. For example, Table 4-1 is a part of an FIT acceptance test table, and Figure 4-1 is a part of the fixture needed. Following the key word "start", "AddInventory" is the name of the fixture class. ‘Check’ is another key word followed by the name of the method that will be invoked when executing the test case and the value 15 is the expected return value from the method. When executing the test case, FIT framework will first automatically converts the name of the method "coffee in inventory" into "coffeeInInventory" by connecting all words and make the initial

characters of those words except the first one into capital, and look for the method named 'coffeeInventory' in the fixture.

fit.ActionFixture		
start	AddInventory	
check	coffee in inventory	15

Table 4-1 Part of AddInventory Test Case

```
import fit.ColumnFixture;
public class AddInventory extends ColumnFixture {
private CoffeeMaker coffeeMaker = new CoffeeMaker();
private Inventory inventory = coffeeMaker.checkInventory();
public int coffeeInInventory() {
return inventory.getCoffee();
}
……
}
```

Figure 4-1 Part of AddInventory Fixture

Since FIT framework executions depend highly on class and method names, a good naming convention is particularly important in FIT. The method name "coffee in inventory" is meaningful and makes the table row read like a human readable sentence. This helps the communication between customers and developers in understanding the test cases in tables. It is also beneficial because customers will be able to fill the FIT tables with test data without much help from the developers. FIT tables contain meaningful class and method names make the work of customers writing acceptance test cases easy. Also, using FIT as business specification document, readable names are crucial for the development and maintenance of the requirement documentation. The table and fixture mentioned above are only a part of the example. The complete table and fixture before refactoring are shown in Table 4-2 and Figure 4-2, and the table and fixture after refactoring are given in Table 4-3 and Figure 4-3.

fit.ActionFixture		
start	AddInventory	
check	coffee inventory	15
check	milk inventory	15
check	sugar inventory	15
check	chocolate inventory	15
enter	units coffee	3
enter	units milk	5
enter	units sugar	6
enter	units chocolate	7
check	coffee inventory	18
check	milk inventory	20
check	sugar inventory	21 expected 15 actual
check	chocolate inventory	22

Table 4-2 AddInventory Table Before Refactoring

fit.ActionFixture		
start	AddInventory	
check	coffee in inventory	15
check	milk in inventory	15
check	sugar in inventory	15
check	chocolate in inventory	15
enter	units coffee to inventory	3
enter	units milk to inventory	5
enter	units sugar to inventory	6
enter	units chocolate to inventory	7
check	coffee in inventory	18
check	milk in inventory	20
check	sugar in inventory	21 expected 15 actual
check	chocolate in inventory	22

Table 4-3 AddInventory Table After Refactoring

```
import fit.ColumnFixture;
public class AddInventory extends ColumnFixture
{    private CoffeeMaker cm = new CoffeeMaker();
     private Inventory i = cm.checkInventory();
     public void unitsCoffee(int coffee) {
          cm.addInventory(coffee,0,0,0);
     }
     public void unitsMilk(int milk) {
          cm.addInventory(0,milk,0,0);
     }
     public void unitsSugar(int sugar) {
          cm.addInventory(0,0,sugar,0);
     }
     public void unitsChocolate(int chocolate) {
          cm.addInventory(0,0,0,chocolate);
     }
     public int coffeeInventory() {
          return i.getCoffee();
     }
     public int milkInventory() {
          return i.getMilk();
     }
     public int sugarInventory() {
          return i.getSugar();
     }
     public int chocolateInventory() {
          return i.getChocolate();
     }
}
```

Figure 4-2 AddInventory Fixture Before Refactoring

```
import fit.ColumnFixture;
public class AddInventory extends ColumnFixture
{    private CoffeeMaker coffeeMaker = new
CoffeeMaker();
     private Inventory inventory =
coffeeMaker.checkInventory();
     public void unitsCoffeeInInventory(int coffee) {
          coffeeMaker.addInventory(coffee,0,0,0);
     }
     public void UnitsMilkInInventory (int milk) {
          coffeeMaker.addInventory(0,milk,0,0);
     }
     public void UnitsSugarInInventory (int sugar) {
          coffeeMaker.addInventory(0,0,sugar,0);
     }
     public void UnitsChocolateInInventory (int
chocolate) {
     coffeeMaker.addInventory(0,0,0,chocolate);
     }
     public int coffeeInInventory() {
          return inventory.getCoffee();
     }
     public int milkInInventory() {
          return inventory.getMilk();
     }
     public int sugarInInventory() {
          return inventory.getSugar();
     }
     public int chocolateInInventory() {
          return inventory.getChocolate();
     }
}
```

Figure 4-3 AddInventory Fixture After Refactoring

In this refactoring example, names of methods and the instances of CoffeeMaker and Inventory are changed to new names that are more meaningful to both developers and customers. The sentence "check coffee in inventory" reads better than "check coffee inventory", and "enter units coffee to inventory" makes more sense than "enter units coffee". Those modifications seem small, but after the refactoring, the tables read more close to the sentences we use in our daily life, which makes the customers more comfortable working with FIT.

4.1.2 Introducing new domain concept

The idea of introducing a new domain concept into FIT tables was first discussed in [9], but the authors did not use it as a refactoring method for FIT acceptance test. A new domain concept is a business concept that is not discovered by customers or not mentioned in the requirement. For example, in geographical information system, when we are looking for a best path from point A to point B with the shortest time, we need to know the length of each section of roads and the speed limit of those sections. But what we really care about is the time we spend on those sections of roads, and the time is directly related to the path with shortest time, not the length and speed limit. In this example, the new domain concept is obvious, but sometimes it is not.

There are several benefits for adding a new domain concept in the current design. Discovering a new domain concept can help developers to understand the nature of the project better. By adding new domain concepts, it also makes the code structure clear, which is good for others to read and understand. Also, it may help to reduce the redundancies in the test cases. By

introducing new domain concept, we manage to break a complex business rule into small ones, which also helps to analyze user stories.

In the next example, the purpose of this system is to offer service packages based on the sales channel, fixed rate and rate adaptive. For fixed rate and rate adaptive, Green means good, Amber means just OK, and Red means bad. The offered services include plan A, B, C, and D.

OfferedServicesFixture			
sales channel	fixed rate	rate adaptive	offered services
web	Green	Green	A, B, C, D
web	Green	Amber	A, B, C
web	Green	Red	A, B, C
web	Amber	Green	A, B, C
web	Amber	Amber	A
web	Amber	Red	

Table 4-4 offteredServices Table

In Table 4-4, we discovered a new domain concept. By analyzing the business requirements, we found that the fixed rate and rate adaptive are actually describing the line capability. So it would be good to make line capability a new domain concept. By introducing the line capability concept, we break the original table into two smaller tables in Table 4-5 and Table 4-6.

LineCapabilityFixture		
fixed rate	rate adaptive	line capability
Green	Green	3
Green	Amber	2
Green	Red	2
Amber	Green	2
Amber	Amber	1
Amber	Red	0

Table 4-5 lineCapability Table

OfferedServicesFixture		
sales channel	line capability	offered services
web	3	A, B, C, D
web	2	A, B, C
web	1	A
web	0	

Table 4-6 The Refactored offeredServices Table

The added domain concept makes the relationship among those data more clear and make the offered services table simpler. The two tables describe a more understandable idea than the original version, and reduce some redundancy. Certainly, the final decision of introducing new domain concepts should be made by the customer.

Another benefit of adding new domain concept is to reduce changes to the test case and code when new business requirements are added. In the following example, when there are additional sales channels added to the system, such as ‘store A’ and ‘store B’, we made many necessary changes as shown in Table 4-7.

offered services fixture			
sales channel	fixed rate	rate adaptive	offered services
web	Green	Green	A, B, C, D
web	Green	Amber	A, B, C
web	Green	Red	A, B, C
web	Amber	Green	A, B, C
web	Amber	Amber	A
web	Amber	Red	
store A	Green	Green	A, B, C, D, E
store A	Green	Amber	A, B, C, E
store A	Green	Red	A, B, C, E
store A	Amber	Green	A, B, C, E
store A	Amber	Amber	A, E
store A	Amber	Red	E
store B	Green	Green	A, B, C, D, E
store B	Green	Amber	A, B, C, E
store B	Green	Red	A, B, C, E
store B	Amber	Green	A, B, C, E
store B	Amber	Amber	A, E
store B	Amber	Red	E

Table 4-7 New offteredServices Table

With the newly added line capability domain concept in our test cases, there is no need to change the line capability fixture table given in Table 4-8. We only need to change offered services fixture table, and there will be fewer changes than the original table without the line capability concept when the new sales channels, store A and store B are added to the system in the future. The most important benefit is that we reduced redundancies in the table. The resulting table "offered services fixture" is showed in Table 4-8.

offered services fixture		
sales channel	line capability	offered services
web	3	A, B, C, D
web	2	A, B, C
web	1	A
web	0	
store A	3	A, B, C, D, E
store A	2	A, B, C, E
store A	1	A, E
store A	0	E
store B	3	A, B, C, D, E
store B	2	A, B, C, E
store B	1	A, E
store B	0	E

Table 4-9 New offeredServices Table After Refactoring

4.1.3 Extract similar part of tables into fixture

When the same steps are repeated in many FIT tables, we can extract the steps into the corresponding fixture code so that they need not be repeated in all test cases.

Before executing acceptance test against the system, sometimes there is the need to set up initial environment so that the acceptance test cases will operate on the same baseline of the system. In the following CoffeeMaker example, the set up means making the default recipes and

after that, the coffee maker machine can sell beverages based on those recipes as seem in Table 4-9. You can see in this table, a large part of the test case is the set up, making the default recipes. If we do this in every test case (FIT table), we will be repeating this default recipe in all test cases, which would be a large redundancy in the test cases. We also do not want those redundancies to appear in the business specification document.

To eliminate the redundancy, we extract the set up steps before the actual test into a new class, and then the corresponding fixtures of the test tables needing the set up will instantiate the new class. Figure 4-4 shows the tables after refactoring.

fit.ActionFixture		
start	coffeemaker.purchaseBeverageNew	
press	start coffee maker	
press	start recipe	
enter	recipe name	Coffee
enter	recipe price	50
enter	recipe coffee units	2
enter	recipe milk units	1
enter	recipe sugar units	1
enter	recipe chocolate units	0
check	add recipe	true
press	start recipe	
enter	recipe name	Latte
enter	recipe price	75
enter	recipe coffee units	2
enter	recipe milk units	2
enter	recipe sugar units	1
enter	recipe chocolate units	0
check	add recipe	true
press	start recipe	
enter	recipe name	Hot Chocolate
enter	recipe price	150
enter	recipe coffee units	0
enter	recipe milk units	1
enter	recipe sugar units	0
enter	recipe chocolate units	2
check	add recipe	true
enter	recipe name	Coffee
check	recipe price	50
enter	user money	55
check	purchase beverage	5
enter	recipe name	Coffee
enter	user money	45
check	purchase beverage	45

Table 4-10 purchaseBeverageNew table

PurchaseBeverage Acceptance Test
The user will not be able to purchase a beverage if they do not deposit enough money into the CoffeeMaker. Change will be dispensed to the user, if they paid more than the price of the beverage. A user's money will be returned if there is not enough inventory to make the beverage.
There are 3 recipes in the CoffeeMaker: Coffee,50,6,1,1,0; Latte, 75,6,6,0,0; and Hot Chocolate,100,0,2,0,2
There are 3 default recipes preseted in the test case.

fit.ActionFixture		
start	coffeemaker.purchaseBeverageNew	
press	start coffee maker	
enter	recipe name	Coffee
check	recipe price	50
enter	user money	55
check	purchase beverage	5
enter	recipe name	Coffee
enter	user money	45
check	purchase beverage	45

Figure 4-4 Refactored purchaseBeverageNew Table

4.1.4 Merge tables

Sometimes business rules use the same or similar set of data to calculate different output. As a result, there will be duplication in the FIT acceptance test. For some of those situations, it would be a good practice to merge those tables together to reduce the duplication.

The following example concerns a company who has many long time customers, and they want to calculate the credit of those customers, and also consider some of the customers VIP's. The company comes up with some business rules to determine the credit and the criteria that qualify customers to be VIP's.

The customer will be allowed a credit up to an amount of $1000.00 if the customer has been trading with the company for more than 12 months, has paid reliably, and has a balance due of less than $6000.00. Otherwise, the customer will have $0 credit with the company. The customer

is considered a VIP if he/she has been trading with the company for more than 12 months and is reliable.

Table 4-10 and Table 4-11 (implementing ColunmFixture) are two FIT acceptance test cases that use the above rules to calculate credit and VIP status of customers.

CalculateCredit				
months	reliable	balance	allow credit()	credit limit()
0	true	0.00	false	0.00
12	true	5500.00	true	1000.00
14	true	5000.00	true	1000.00
18	true	6000.00	false	0.00
24	false	0.00	false	0.00

Table 4-11 CalculateCredit table

CalculateVIP		
months	reliable	VIP()
0	true	false
12	true	true
14	true	true
18	true	true
24	false	false

Table 4-12 CalculateVIP Table

Observe that all the data in CalculateVIP table, months and reliable, appear in the CalculateCredit table. Those two set of data are duplicated, and it is desirable to eliminate this duplication. We can merge them into a larger table, and of course, we need to change the corresponding fixture code. The change of the fixture code should be easy. The merged table is listed in Table 4-12.

CalculateCredit					
months	reliable	balance	allow credit()	credit limit()	VIP()
0	true	0.00	false	0.00	false
12	true	5500.00	true	1000.00	true
14	true	5000.00	true	1000.00	true
18	true	6000.00	false	0.00	true
24	false	0.00	false	0.00	false

Table 4-13 CalculateCredit Table

4.1.5 Split table

On the other hand, there are times we need to break a large table into smaller pieces. For example, if the two sets of criteria used in the previous example for merging tables in Section 5.1.4 are more different than they were, then the two original tables should not be merged. If, for whatever reasons, the two set of different criteria were tested in the same table, we should consider breaking them into two tables. The following is another example:

Rule to calculate credit - for a customer who has been trading with the company for more than 12 months, has paid reliably over that period, and has a balance due of less than $6000.00, can have credit up to $1000.00; for a customer who has been trading with the company for more than 24 months, has paid reliably over that period, and has a balance due of less than $4000.00, can have credit up to $2000.00; for a customer who has been trading with the company for more than 30 months, has paid reliably over that period, and has a balance due of less than $4000.00, can have credit up to $3000.00. Rule to decide VIPs: if the customer has been trading with the company for more than 12 months and reliable, then he is a VIP.

A FIT acceptance test case before refactoring is given in Table 4-13. In this large table, it is apparent that most of the test rules are not useful for the calculation of VIP status, so they are

irrelevant. And in such a table, the rule to calculate VIP can not be identified easily, which is another drawback of using such a large table. So we broke the table into two smaller ones in Table 4-14 and Table 4-15.

And the corresponding fixture needs to be split into two smaller fixtures as well. Although the two test tables look like more than what we had before, the calculation of VIP is more clearly presented in the table CalculateVIP.

CalculateCredit					
months	reliable	balance	allow credit()	credit limit()	VIP()
0	true	0.00	false	0.00	false
12	true	5500.00	true	1000.00	true
12	true	6000.00	false	0.00	true
12	true	7000.00	false	0.00	true
14	true	5000.00	true	1000.00	true
18	true	6000.00	false	0.00	true
18	false	4000.00	false	0.00	false
24	false	0.00	false	0.00	false
24	true	3000.00	true	2000.00	true
24	true	4000.00	false	0.00	true
29	true	2000.00	true	2000.00	true
30	false	2000.00	false	0.00	false
30	true	3000.00	true	3000.00	true
30	true	4000.00	false	0.00	true
31	true	3999.00	true	3000.00	true

Table 4-14 The New CalculateCredit Table

CalculateCredit				
months	reliable	balance	allow credit()	credit limit()
0	true	0.00	false	0.00
12	true	5500.00	true	1000.00
12	true	6000.00	false	0.00
12	true	7000.00	false	0.00
14	true	5000.00	true	1000.00
18	true	6000.00	false	0.00
18	false	4000.00	false	0.00
24	false	0.00	false	0.00
24	true	3000.00	true	2000.00
24	true	4000.00	false	0.00
29	true	2000.00	true	2000.00
30	false	2000.00	false	0.00
30	true	3000.00	true	3000.00
30	true	4000.00	false	0.00
31	true	3999.00	true	3000.00

Table 4-15 Refactored New CalculateCredit Table

CalculateVIP		
months	reliable	VIP()
0	true	false
12	true	true
14	true	true
18	true	true
24	false	false

Table 4-16 Refactored New CalculateVIP Table

4.2 FIT refactoring methods on fixtures

Refactoring on fixtures is very similar to refactoring on system code though there are some differences. FIT fixtures are relatively simple compared to system code. What we usually do in fixtures is to call the functions in system code, so the fixtures should not be "fat", which means they should not contain too much business logic. The business logic should be in the system code but not acceptance test fixtures. So, for those refactoring methods related to complicated class inheriting, they might not be suitable for refactoring on FIT fixtures. In the following sections, we introduce several methods.

4.2.1 Merge similar methods

In the FIT acceptance test, there are times you may work on a set of same type of items. Those items are simple, but you have to have corresponding set of methods to operate them. A

good example is the CoffeeMaker system; you have coffee, milk, sugar and chocolate as the beverage ingredient. You need methods to set the amount of the ingredient in the coffee maker machine. In the system under test, we have a list of such methods: *setAmtCoffee(int coffee), setAmtMilk(int milk), setAmtSugar(int sugar) and setAmtChocolate(int chocolate)*. Here we show a part of the FIT acceptance test in Table 4-16, which tests the add recipe function. And the corresponding fixture is showed in Figure 4-5.

enter	recipe to edit	Coffee
check	edit recipe	false
press	start recipe	
enter	recipe name	Coffee
enter	recipe price	50
enter	recipe coffee units	2
enter	recipe milk units	1
enter	recipe sugar units	1
enter	recipe salt units	1
enter	recipe honey units	0
enter	recipe chocolate units	0
check	add recipe	true

Table 4-17 Part of EditRecipe Before Refactoring

enter	recipe to edit	Coffee
check	edit recipe	false
press	start recipe	
enter	recipe name	Coffee
enter	recipe price	50
enter	Recipe item	coffee
enter	Item amount	2
enter	Recipe item	milk
enter	Item amount	1
enter	Recipe item	salt
enter	Item amount	0
enter	Recipe item	honey
enter	Item amount	0
enter	Recipe item	chocolate
enter	Item amount	0
check	add recipe	true

Table 4-18 Part of EditRecipe After Refactoring

```
import fit.ActionFixture;
public class AddRecipe extends ActionFixture {
    Recipe recipe;
    CoffeeMaker cm = new CoffeeMaker();
    public void startRecipe() {
            recipe = new Recipe();
    }
    public void recipeName(String name) {
            recipe.setName(name);
    }
    public void recipePrice(int price) {
            recipe.setPrice(price);
    }
    public void recipeCoffeeUnits(int coffee) {
            recipe.setAmtCoffee(coffee);
    }
    public void recipeMilkUnits(int milk) {
            recipe.setAmtMilk(milk);
    }
    public void recipeSugarUnits(int sugar) {
            recipe.setAmtSugar(sugar);
    }
    public void recipeSaltUnits(int salt)    {
            recipe.setAmtSalt(salt);
    }
    public void recipe Honey Units(int honey)    {
            recipe.setAmt Honey (honey);
    }
    public void recipeChocolateUnits(int
chocolate)    {
            recipe.setAmtChocolate(chocolate);
    }
    public boolean addRecipe() {
            return cm.addRecipe(recipe);
    }
}
```

Figure 4-5 EditRecipe Fixture Before Refactoring

```
    Public string ItemInRecipe;
    Public void recipeItem(string whatWeNeed ){
            ItemInRecipe = whatWeNeed;
}
Public void itemAmount(int amt){
    If(ItemInRecipe.equals("coffee"))
            oldRecipe.setAmtCoffee(amt);
    else If(ItemInRecipe.equals("milk"))
            oldRecipe.setAmtMilk(amt);
    else If(ItemInRecipe.equals("sugar"))
            oldRecipe.setAmtSugar(amt);
    else If(ItemInRecipe.equals("salt"))
            oldRecipe.setAmtSalt(amt);
    else If(ItemInRecipe.equals("honey"))
            oldRecipe.setAmtHoney(amt);
    else If(ItemInRecipe.equals("chocolate"))
            oldRecipe.setAmtChocolate (amt);
}
```

Figure 4-6 EditRecipe Fixture Before Refactoring

There are four functions to operate on each of the four ingredients to make a beverage, and we need to list them one by one in the fixture. If there are more than four ingredients, fourteen for instance, it would be a disaster to list them all in the fixture and it would be hard for customers and developers to remember all of the ingredients. Such a long list of similar methods would be a bad design in the acceptance test. We should consider extracting those methods into one. However, there is one problem with extracting them into one method because ActionFixture in FIT framework can take only one parameter for one method [19]. We can’t manage to make them into exactly one method since we need the names of the ingredients and the quantities to

add to the coffee machine. One solution is to make them into two methods such as the fixture given in Figure 4-6, and the corresponding FIT test case table is given in Table 4-17. This way, the fixture gets rid of the long list of similar methods, and customers and developers do not need to remember those method names. If the requirements change and there are new items added to the system, we can simply modify the existing methods to accommodate the changes without creating new methods, which makes the test code cleaner. The only minor drawback is that the fixture becomes longer with the additional "else if" conditions. The test case seems to be longer than the original, but it looks more readable and more understandable. We reduce the number of methods which have limited usages, and added two universal methods to do the job.

This refactoring method may not be suitable for all situations. If the number of those similar methods in the fixture is relatively small and is not likely to increase, then it would be unnecessary to combine those smaller methods into one or two general methods.

4.2.2 Extract method

Extract Method is one of the most important refactoring methods in [7], which is also quite suitable for FIT refactoring. In the fixture code, if some pieces of code are always executing together and are logically related, then it would be wise to extract them into a method and give the new method a meaningful name. There are several benefits for doing this: the method can be reused, so when you want to use that block of code again, you just call the method; the structure is more understandable, and the methods are smaller; with a good name, when others read the code, they don't need to go into the detail of the new method.

The following introduces an ActionFixture in the CoffeeMaker example. In the ActionFixture, before testing the purchase beverage function, we need to set up the initial condition by adding recipes (makeRecipes() method). The original fixture doing this is in Figure 4-7. It can be seen that in the makeRecepies() method, the same actions of "creating new recipe" and "add to the machines" are repeated several times. We can extract the actions into one method, named makeRecipe() which is called from makeRecipes(). The refactored fixture is shown is Figure 4-8. Compare with the ActionFixture before, this refactored one is short, well-structured, easy to understand.

```
import fit.ActionFixture;
public class PurchaseBeverage extends
ActionFixture {      private Recipe recipe;
      private Recipe r1, Recipe r2, Recipe r3;
      private int money;
      private CoffeeMaker cm = new CoffeeMaker();
      public void startCoffeeMaker() {
            makeRecipes();       }
      public void recipeName(String name) {
            this.recipe =
cm.getRecipeForName(name);}
      public void userMoney(String money) {
            this.money = Integer.parseInt(money);      }
      public int purchaseBeverage() {
            return cm.makeCoffee(recipe,money);      }
      private void makeRecipes() {
            r1 = new Recipe();
            r1.setName("Coffee");
            r1.setPrice(50);
            r1.setAmtCoffee(6);
            r1.setAmtMilk(1);
            r1.setAmtSugar(1);
            r1.setAmtChocolate(0);
            r2 = new Recipe();
            r2.setName("Latte");
            r2.setPrice(75);
            r2.setAmtCoffee(6);
            r2.setAmtMilk(6);
            r2.setAmtSugar(0);
            r2.setAmtChocolate(0);
            r3 = new Recipe();
            r3.setName("Hot Chocolate");
            r3.setPrice(100);
            r3.setAmtCoffee(0);
            r3.setAmtMilk(2);
            r3.setAmtSugar(0);
            r3.setAmtChocolate(2);
            cm.addRecipe(r1);
            cm.addRecipe(r2);
            cm.addRecipe(r3);
      }
}
```

Figure 4-7 PurchaseBeverage Before Refactoring

```
import fit.ActionFixture;
public class PurchaseBeverage extends
ActionFixture {
      private Recipe recipe;
      private Recipe recipe1, recipe2, recipe3;
private int moneyPaid;
      private CoffeeMaker cm = new CoffeeMaker();
      public void startCoffeeMaker() {
            makeRecipes();}
      public void recipeName(String name) {
            this.recipe =
cm.getRecipeForName(name);      }
      public void userMoney(String money) {
            this.moneyPaid = Integer.parseInt(money);
      }
      public int purchaseBeverage() {
            return cm.makeCoffee(recipe,moneyPaid);
      }
      private void makeRecipes() {
            //Coffee,50,6,1,1,0; Latte, 75,6,6,0,0; and
Hot Chocolate,100,0,2,0,2
            makeRecipe(recipe1, “Coffee”, 50, 6, 1, 1,
0);
            makeRecipe(recipe2, “Latte”, 75, 6, 6, 0,
0);
            makeRecipe(recipe3, “Hot Chocolate”,
100, 0, 2, 0, 2);
            cm.addRecipe(recipe1);
            cm.addRecipe(recipe2);
            cm.addRecipe(recipe3);
      }
  private void makeRecipe(Recipe recipe, String
name, int price, int coffee, int milk, int sugar, int
chocolate) {
        recipe = new Recipe();
      recipe.setName(name);
      recipe.setPrice(price);
      recipe.setAmtCoffee(coffee);
      recipe.setAmtMilk(milk);
      recipe.setAmtSugar(sugar);
      recipe.setAmtChocolate(chocolate);
  }
}
```

Figure 4-8 PurchaseBeverage After Refactoring

4.2.3 Extract similar methods in fixture into new class

Sometimes, in FIT acceptance test cases, there are similar methods in many fixtures. The most common situation is again setting up the initial testing environment, which means that you

need to set some variables in the system as default values, and also, after one test case, you need to set those variables back to the default values, so that there is no carry-over effect on the next test case. In some project, when we are doing acceptance test, those set up steps may be repeated many times, it would beneficial if there is a set up class, and we can call it when needed.

Using the CoffeeMaker example, the system is the software running as a coffee maker machine. The customer of the project should be able to add recipes like coffee, latte or chocolate, and users of the machine can buy those beverages. When testing the purchase beverage function, or the edit or delete recipe functions, we need some recipes prepared in the system. We need to set the name and price for the recipe, and also the amount of sugar, coffee and milk in the recipe.

```
import fit.ActionFixture;
public class PurchaseBeverage extends ActionFixture {
    private Recipe recipe;
    private int money;
    private CoffeeMaker cm = new CoffeeMaker();
    public void startCoffeeMaker() {
        makeRecipes();
    }
    public void recipeName(String name) {
        this.recipe = cm.getRecipeForName(name);
    }
    public void userMoney(String money) {
        this.money = Integer.parseInt(money);
    }
    public int purchaseBeverage() {
        return cm.makeCoffee(recipe,money);
    }
    public void makeRecipes() {
        //Coffee,50,6,1,1,0; Latte, 75,6,6,0,0; and Hot Chocolate,100,0,2,0,2
        makeRecipe("Coffee", 50, 6, 1, 1, 0);
        makeRecipe("Latte", 75, 6, 6, 0, 0);
        makeRecipe("HotChocolate", 100, 0, 2, 0, 2);
    }
    public void makeRecipe(String name, int price, int coffee, int milk, int sugar, int chocolate) {
        Recipe recipe = new Recipe();
        recipe.setName(name);
        recipe.setPrice(price);
        recipe.setAmtCoffee(coffee);
        recipe.setAmtMilk(milk);
        recipe.setAmtSugar(sugar);
        recipe.setAmtChocolate(chocolate);
        cm.addRecipe(recipe);
    }
}
```

Figure 4-9 PurchaseBeverage Before Refactoring

Quite a few places in the acceptance test may need exactly the same set up, so a set up class would be helpful. We can create a method class to fulfill this work. The class in Figure 4-10 is to set up the commonly used environment. Now we have the class, setDefaultRecipes to set up the environment, we can use it directly. The fixture in Figure 4-11 calls the set up class in Figure 4-10.

```
public class setDefaultRecipes{
    private CoffeeMaker coffeeMaker;
    public setDefaultRecipes(){
        coffeeMaker = new CoffeeMaker();
        makeRecipes();
    }
    public CoffeeMaker returnDefault(){
        return coffeeMaker;
    }
    public void makeRecipes() {
        //Coffee,50,6,1,1,0; Latte, 75,6,6,0,0;
and Hot Chocolate,100,0,2,0,2
        makeRecipe("Coffee", 50, 6, 1, 1, 0);
        makeRecipe("Latte", 75, 6, 6, 0, 0);
        makeRecipe("HotChocolate", 100, 0,
2, 0, 2);
    }
    public void makeRecipe(String name, int
price, int coffee, int milk, int sugar, int
chocolate) {
        Recipe recipe = new Recipe();
        recipe.setName(name);
        recipe.setPrice(price);
        recipe.setAmtCoffee(coffee);
        recipe.setAmtMilk(milk);
        recipe.setAmtSugar(sugar);
        recipe.setAmtChocolate(chocolate);
        coffeeMaker.addRecipe(recipe);
    }
}
```

Figure 4-10 setDefaultRecipes Class

```
import fit.ActionFixture;
public class purchaseBeverageNew extends
ActionFixture {
    private Recipe recipe;
    private int money;
    private CoffeeMaker cm = new CoffeeMaker();
    private setDefaultRecipes setDefault;
    public void startCoffeeMaker() {
        setDefault = new setDefaultRecipes();
        cm = setDefault.returnDefault();
    }
    public void recipeName(String name) {
        this.recipe =
cm.getRecipeForName(name);
    }
    public int recipePrice(){
        return this.recipe.getPrice();
    }
    public void userMoney(String money) {
        this.money = Integer.parseInt(money);
    }
    public int purchaseBeverage() {
        return cm.makeCoffee(recipe,money);
    }
}
```

Figure 4-11 purchaseBeverageNew Fixture

The fixtures set the default values in the program, so the customers can't know it visually. It would be wise to introduce those values in the test case. Sometimes displaying all the set up details would take too much space and would be redundant to explain every time, so we may

need a page to explain the details and have a link on the teat case page linking to that page.

Figure 4-12 shows the test table.

PurchaseBeverage Acceptance Test
The user will not be able to purchase a beverage if they do not deposit enough money into the CoffeeMaker. Change will be dispensed to the user, if they paid more than the price of the beverage. A user's money will be returned if there is not enough inventory to make the beverage.
There are 3 recipes in the CoffeeMaker: Coffee,50,6,1,1,0; Latte, 75,6,6,0,0; and Hot Chocolate,100,0,2,0,2
There are 3 default recipes preseted in the test case.

fit.ActionFixture		
start	coffeemaker.purchaseBeverageNew	
press	start coffee maker	
enter	recipe name	Coffee
check	recipe price	50
enter	user money	55
check	purchase beverage	5
enter	recipe name	Coffee
enter	user money	45
check	purchase beverage	45

Figure 4-12 PurchaseBeverage Test Case

CHAPTER 5 OTHER FINDINGS

In the process of working on the FIT refactoring methods, we found that the smells in acceptance test sometimes indicate smells in the system code. This can be useful in case that the programmer did not find the smell or there was just no time to review that. We also give some recommendations on how to organize a FIT table, and this should help to make the test cases better organized.

5.1 Improper FIT acceptance test may indicate improper system code

In FIT acceptance test, the fixtures need to instantiate the classes in the system under test, and call the methods in the classes. There are times that we found smells in fixtures, like the name for a method is not presenting what the method is really doing, or one method is doing too much. And sometimes those smells are caused by the methods in the system under test, but not the methods in the fixture. So when we want to do refactoring on the fixture code, we need to modify the system code, which is no longer refactoring on acceptance test. This means the smells in FIT acceptance test may indicate smells in system code.

The smells in the system code should be found earlier. There are chances for them to be noticed: just after finishing the method code, while doing unit test or in the code review. However, in the step of refactoring on acceptance test is also a possible time. There might be reasons for them to escape from people's sight in the other steps, but it is nice we finally find them.

The following are two examples in which you find some smells in the fixtures or in the test case tables, and when you try to refactor, you find the same smells in the system code. The smell in FIT test indicates the smell in system under test.

The first example is a coffeeMaker class, and two of the member methods are makeRecipe() and updateRecipe(). As you can see in the class in Figure 5-1, both of them have a long list of parameters. It is obvious that using such a long list of parameters is a smell in most cases, but you may not discover this until you are reviewing your FIT acceptance test cases. In the fixture, you will instantiate the coffeeMaker object and call the methods like the code in the fixture in Figure 5-2.

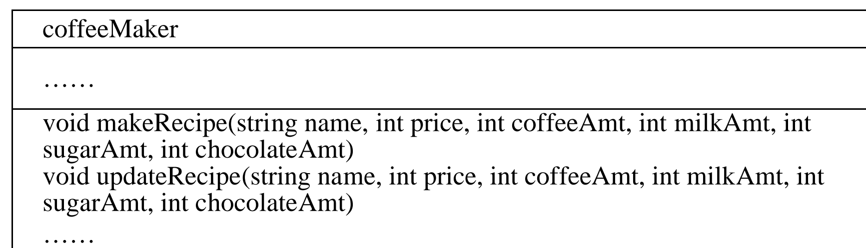


Figure 5-1 coffeeMaker Class Diagram

```
import fit.ActionFixture;
public class setUpCoffeeMaker extends ActionFixture {
……
    private CoffeeMaker cm = new CoffeeMaker();
    ……
    cm. makeRecipe(recipeName, price, coffeeAmt, milkAmt, sugarAmt, chocolateAmt)
    ……
    cm. updateRecipe(recipeName, price, coffeeAmt, milkAmt, sugarAmt, chocolateAmt)
    ……
}
```

Figure 5-2 Part of setUpCoffeeMaker Fixture

It is considered a mess to call a method with a large bunch of logically related parameters. A data class named recipe containing all the information about the recipe should be introduced.

The methods makeRecipe() and updateRecipe() can use the new data class as parameter. That is the refactoring method in [7], Introduce Parameter Object. Further, you find those two methods should be in the class recipe but not in the class coffeeMaker. Then you move the methods into the new class. That is the refactoring called Move Method in [7]. Now you are changing the system code, and that is not FIT refactoring, but it is the refactoring on acceptance that helps to discover the smell in system code.

Here is the second example. You have a class hierarchy, superclass “Employee” and subclass “Manager”. In your FIT tests, you somehow find that the subclass’s own data structure or methods are not frequently used, or maybe they are seldom used. The worst case is that the subclass does not even have its own data or method, exactly the same with the super class. Then you may want to go to the system code and do the Collapse Hierarchy [7] refactoring on it. That is another case you find a smell in the acceptance test and it indicates a smell in the system code.

Actually, these refactorings can be done when doing refactoring on the system code. But if we did not find the smells when doing system code refactoring, refactoring on acceptance test will help us to find them. It gives another chance to revise mistakes and make the system better. Especially when the code is written and tested by some non-experienced programmer, this would be helpful.

5.2 Recommendation on organization of FIT acceptance test table

For a large project, there would be huge amount of acceptance test cases. So how to organize the test cases would be an important issue. If you are using FIT as the acceptance test

tool, then you have many tables. Here we give a recommendation on how to organize the FIT acceptance test tables.

If you Use FIT as the acceptance test tool, tables are used to contain the test cases, and the issue of how to organize tables comes to mind. The purposes of this section are to organize the data better, to give a clear view of test cases, and make tables in HTML, MS Excel, MS Word or some other application more friendly to customers. So, the organization of tables should make it easy for customers to work on and clear to read and understand. For FIT, you can put several test cases into one table, or you can make one test case in several small tables, but sharing the same fixture. Which one would be good, long table containing test cases or short table with part of a test case?

Use a huge table containing several test cases has defects. A long table containing hundred or maybe thousand entries would be annoying to read. Customers will work on the table and fill the acceptance test case with data, and the long table would seem endless for them. Maybe when the customers comes to the half of the table, and totally forget what he or she has read before, and have to go back from the beginning of the table. It will keep customers away from working pleasantly with testers. When used as business specification document, huge sections would also be boring to read.

fit.ActionFixture		
start	coffeemaker.AddRecipe	
press	start recipe	
enter	recipe name	Coffee
enter	recipe price	50
enter	recipe coffee units	2
enter	recipe milk units	1
enter	recipe sugar units	1
enter	recipe chocolate units	0
check	add recipe	true
press	start recipe	
enter	recipe name	Latte
enter	recipe price	75
enter	recipe coffee units	2
enter	recipe milk units	2
enter	recipe sugar units	1
enter	recipe chocolate units	0
check	add recipe	true
press	start recipe	
enter	recipe name	Hot Chocolate
enter	recipe price	150
enter	recipe coffee units	0
enter	recipe milk units	1
enter	recipe sugar units	0
enter	recipe chocolate units	2
check	add recipe	true
press	start recipe	
enter	recipe name	Mocha
enter	recipe price	50
enter	recipe coffee units	2
enter	recipe milk units	0
enter	recipe sugar units	1
enter	recipe chocolate units	1
check	add recipe	false expected true actual

Table 5-1 coffeemaker.AddRecipe Table Before Refactoring

There are two benefits of using small tables. First, the small tables look friendly; they are not annoying, easier to read. Second, you can insert comments between tables, explaining what each small table is doing. If you use FIT as the business specification document, this would be great because it makes the document easy to read and clear.

For example, we have a long table in the coffeeMaker project, and this table adds recipes to the coffee maker machine. There are several recipes need to be added, so the test cases need to repeat the similar action for several times. You can consider they are separate test cases, so the long table contains more than one test case. This table is a large one, and if we add some more recipes, it would be longer. The table in Table 5-1 does not look good and lacks structure. We can break the table into some smaller one, and they are still using the same fixture. We also add some comments between tables, making them more understandable. The revised tables are listed in Table 5-2, Table 5-3, Table 5-4 and Table 5-5.

fit.ActionFixture		
start	AddRecipe	
press	start recipe	
enter	recipe name	Coffee
enter	recipe price	50
enter	recipe coffee units	2
enter	recipe milk units	1
enter	recipe sugar units	1
enter	recipe chocolate units	0
check	add recipe	true

The above table starts the AddRecipe fixture and adds the recipe named coffee.

Table 5-2 New AddRecipe Part 1

fit.ActionFixture		
press	start recipe	
enter	recipe name	Latte
enter	recipe price	75
enter	recipe coffee units	2
enter	recipe milk units	2
enter	recipe sugar units	1
enter	recipe chocolate units	0
check	add recipe	true

The above table adds the recipe named latte.

Table 5-3 New AddRecipe Part 2

fit.ActionFixture		
press	start recipe	
enter	recipe name	Hot Chocolate
enter	recipe price	150
enter	recipe coffee units	0
enter	recipe milk units	1
enter	recipe sugar units	0
enter	recipe chocolate units	2
check	add recipe	true

The above table adds the recipe named hot Chocolate.

Table 5-4 New AddRecipe Part 3

fit.ActionFixture		
press	start recipe	
enter	recipe name	Mocha
enter	recipe price	50
enter	recipe coffee units	2
enter	recipe milk units	0
enter	recipe sugar units	1
enter	recipe chocolate units	1
check	add recipe	false expected true actual

The above table adds the recipe named Mocha.

Table 5-5 New AddRecipe Part 4

CHAPTER 6 CONCLUSION

In this research, we introduce the concept of FIT Refactoring and showed that it is necessary and beneficial. Although the ideas of organizing test cases have been discussed previously, there is no prior research on applying refactoring to FIT acceptance test. We have also introduced FIT table refactoring which is quite different from the concept of code refactoring. Several FIT Refactoring methods are provided in this research and are shown easy to use.

We have found that FIT Refactoring is necessary, important, and easy to apply. FIT acceptance test is important in many aspects of a software development project. It automates the acceptance testing process, facilitates the communication between customers and developers, and serves as requirement specification documentations. So it is necessary to maintain the good structures of all test cases. But there are times that changes may cause test cases structures to be deviated from a good design. For instance, when the size of the system grows, the number of test cases may also grow rapidly. The large number of test cases might make the original design no longer desirable, so a good design becomes bad. Another example could be that when new features are being added to the system, FIT acceptance test needs to be changed accordingly, so the good design structure is destroyed. In those cases, to maintain the good design of the FIT test tables and fixtures, FIT refactoring becomes necessary for projects using FIT.

FIT Refactoring is easy to do since the concept and the techniques of refactoring are often inline with the principles of a good programming. FIT Refactoring is relatively simple compare

to the refactoring on system code because in acceptance test there are no complicated logics and complex class hierarchies. Thus there is no need for complex refactoring methods in FIT acceptance test. Furthermore, many IDEs (Integrated Development Environment), such as Eclipse, have built-in tools for code refactoring, which can also be used on fixture refactoring. For FIT table refactoring, since the tables could be created in HTML, Microsoft Word, Excel or Notepad which are easy and available tools, working with tables is relatively easy.

It should also be noted that the table refactoring is similar in concept to the restructuring of tables in a relational database. For example, when designing a table in a relational database, a meaningful name is as important as it is in FIT tables, so the "Rename" method can be used in both database tables and FIT tables. Another example is the proposed FIT refactoring method "Introduce New Domain Concept" which is similar to the table normalization in database design; we attempt to eliminate anomalies in tables.

Due to the limitation of time and resource for this research, the FIT refactoring methods created in this research are not enough to handle all situations in generating FIT acceptance test cases. More FIT refactoring methods are needed in the future. While there are tools for automating refactoring on system code, which can also be used for FIT fixture refactoring, more of these tools are needed, and tools for automating FIT table refactoring are also in greater needs.

REFERENCES

[1] Kent Beck, Extreme Programming Explained: Embrace Change, published by Addison-Wesley, ISBN 0-201-61641-6

[2] Eric M. Burke, Brian M. Coyner, Java Extreme Programming Cookbook, O'Reilly, ISBN: 0-596-00387-0, 2003

[3] Arie van Deursen, Leon Moonen, Alex van den Bergh and Gerard Kok, "Refactoring Test Code", Proc. second International Conference on eXtreme Programming and Flexible Processes in Software Engineering XP2001.

[4] Chao, J. (2007). Student project collaboration using Wikis. Proceedings of the 20th Conference on Software Engineering Education and Training (CSEE&T 2007), Dublin, Ireland: July 3-5.

[5] Carl Erickson, Ralph Palmer, David Crosby, Michael Marsiglia1 and Micah Alles, "Make Haste, Not Waste: Automated System Testing", Extreme Programming and Agile Methods - XP/Agile Universe 2003, page 120-128, 2003.

[6] Martin Fowler, Mocks Aren't Stubs, last update Jan 2nd 2007, http://martinfowler.com/articles/mocksArentStubs.html

[7] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts,"Refactoring: Improving the Design of Existing Code", published by Addison-Wesley Professional, 1999, ISBN-10: 0201485672.

[8] Opdyke, W. F. 1992, "Refactoring object-oriented frameworks", Ph.D. thesis, Urbana-Champaign, IL, USA.

[9] David Gallardo, Refactoring for everyone, online, , last update Sep 9th 2003, http://www-128.ibm.com/developerworks/library/os-ecref/

[10] Prashant Gandhi, Nils C. Haugen, Mike Hill, Richard Watt, "Creating a living specification using FIT documents", Proceedings of the Agile Development, Pages: 253 – 258, 2005

[11] Grigori Melnik, Kris Read, Frank Maurer, "Suitability of FIT User Acceptance Test for Specifying Functional Requirements: Developer Perspective", Proc. XP/Agile Universe 2004, Lecture Notes in Computer Science, Vol. 3134, Springer Verlag: 60-72, 2004.

[12] Gerard Meszaros, xUnit Test Patterns: Refactoring Test Code, Addison-Wesley Professional, ISBN: 978-0131495050, 2007

[13] Gerard Meszaros, Shaun M. Smith and Jennitta Andrea, "The Test Automation Manifesto", Proc. Extreme Programming and Agile Methods - XP/Agile Universe 2003, Lecture Notes in Computer Science, Volume 2753/2003, Sep 5th 2003

[14] Rick Mugridge, Ward Cunningham, "Fit for Developing Software: Framework for Integrated Tests", published by Prentice Hall, ISBN-10: 0-321-26934-9.

[15] Kris Read, Grigori Melnik, Frank Maurer, "Examining Usage Patterns of the FIT Acceptance Testing Framework", Proc. 6th International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP2005), Lecture Notes in Computer Science, Vol. 3556, Springer Verlag: 127-136 2005.

[16] Kris Read, Grigori Melnik, Frank Maurer, "Student Experiences with Executable Acceptance Testing", Proc. Agile 2005 Conference, IEEE Press: 312-317, 2005.

[17] Rashmi Singhal, Amelio Vazquez-Reina, Songwei Tsay, Chris Palmer, and David Dennis, "Refactoring", Department of Computer Science, Tufts University.

[18] Ken Schwaber and Mike Beedle, Agile Software Development with Scrum, published by Prentice Hall, 2001, ISBN 0130676349

[19]FIT website, Introduction To Fit, online, Last edited March 1, 2005, http://fit.c2.com/wiki.cgi?IntroductionToFit.

[20] Wikipedia, Refactoring, online, last update February 2nd 2007 http://en.wikipedia.org/wiki/Refactoring

[21] NCSU, Acceptance Testing in Eclipse Using FIT, online, last update August 29th 2005, http://open.ncsu.edu/se/tutorials/fit

[22] Agile Alliance, What Is Agile Software Development? Online, last update May 14th 2006, http://www.agilealliance.org/show/2

[23] UIUC, Refactoring Browser, Online, last update Jun 18th 1999, http://st-www.cs.uiuc.edu/users/brant/Refactory/

[24] Jetbrians, ReSharper, Online, http://www.jetbrains.com/resharper/features/refactoring.html