hochschule mannheim

Fakultät Informatik

Master-Thesis

Refactoring of Acceptance Tests

Vorgelegt von:	Heiko Ordelt xxx xxx hordelt@gmail.com
Matrikelnummer:	xxx
Zeitraum:	01.10.2007 – 31.03.2008
Erstgutachterin:	Prof. Dr. Astrid Schmücker-Schend
Zweitgutachter:	Prof. Dr. Peter Knauber
Praktischer Teil angefertigt bei:	Prof. Dr. Frank Maurer Agile Software Engineering Group University of Calgary Department of Computer Science 2500 University Dr NW Calgary, Alberta T2N 1N4 Canada

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit hat in dieser oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegen.

Mannheim, 31.03.2008 _____

Unterschrift

Abstract

Executable Acceptance Test Driven Development (EATDD) is used to perform the test-first paradigm on customer level. In EATDD, requirements of the system are first translated into business-facing executable acceptance tests before developers start to work on that particular feature. It provides the customer with the confidence that the system satisfies his expectations and helps developers to understand the requirements better.

Since requirements can change over time, the appropriate acceptance tests have to be altered to be up-to-date with the customers' expectations. This process can be time-consuming and risky as inconsistencies can be overlooked easily. Additionally, acceptance tests have to be modified to improve their readability and understandability.

Refactoring of acceptance tests is used to keep fixture and acceptance test definition consistent. The automated refactoring support for FitClipse allows the user to carry out changes to acceptance tests in an efficient and safe manner. It decreases the risk of faults and helps keeping the tests up-to-date.

German Abstract

In Executable Acceptance Test Driven Development (EATDD) wird das "Test-First"-Paradigma auf die Kundenebene angewendet. Dies erfolgt durch Übersetzen der Anforderungen eines Software-Systems in geschäftsorientierte ausführbare Akzeptanztests. Die Entwicklung einer bestimmten Funktionalität beginnt erst, nachdem der dazugehörige Akzeptanztest erstellt wurde. Dies gibt dem Kunden die Gewissheit, dass das entwickelte System seine Erwartungen erfüllt und hilft dem Entwicklungsteam die Anforderungen besser zu verstehen.

Da Systemanforderungen während der Entwicklung geändert werden können, müssen die entsprechenden Akzeptanztests bei Bedarf angepasst werden um die Erwartungen des Kunden widerzuspiegeln. Dieser Prozess kann sehr zeitaufwendig und fehleranfällig sein, da Inkonsistenzen zwischen Test-Definition und Fixture leicht übersehen werden können. Des Weiteren müssen Akzeptanztests modifiziert werden um die Lesbarkeit und Verständlichkeit zu verbessern.

Refactoring von Akzeptanztests hält Test-Definition und die dazugehörigen Fixtures konsistent. Die für diese Arbeit entwickelte automatisierte Refactoring-Funktion in FitClipse erlaubt es dem Benutzer Änderungen an Akzeptanztests effizient und sicher auszuführen. Diese Unterstützung verringert das Fehlerrisiko und hilft die Tests effektiv und aktuell zu halten.

Publications

Content, ideas and figures from this thesis have appeared previously in the following publication:

Heiko Ordelt, Frank Maurer: **Acceptance Test Refactoring**, *Proceedings 9th International Conference on Agile Processes and eXtreme Programming in Software Engineering (XP2008), Limerick, Ireland, Springer, 10-14 June 2008.*

Acknowledgments

Without the help of many people, I would not been able to create this work. I would like to thank these people by heart:

At first, Prof. Dr. Schmücker-Schend and Prof. Dr. Frank Maurer for supervising my thesis and making my stay in Calgary possible as well as Prof. Dr. Peter Knauber for being my co-supervisor and giving me the opportunity to join the master course.

The members of the ASE group of the University of Calgary, Johannes Fischer and Felix Riegger for all the support and the great time we spent together.

Last but not least, I would like to use the opportunity to thank everyone who has helped me during my studies. Your support has significantly contributed to my success.

Dedication

It was a long way.

To all people who have been supporting me over the last years.

Table of Figures

List of Tables

Table of Contents

1 Introduction

1.1 Executable Acceptance Test-Driven Development

Test Driven Development (TDD) is a well-known software development technique that follows the test-first approach. In TDD, developers write test code before any production code is written. This among other aspects leads to better designed code and gives a regression safety net which helps to find bugs quicker.

While TDD works on a level of methods and functions, Executable Acceptance Test Driven Development (also called story tests (Kerievsky, 2004), customer tests (Beck, et al., 2004), example driven development (Marick, 2003) and scenario tests (Kaner, 2003)) pushes the TDD paradigm up to the level of features or requirements. Executable Acceptance Test Driven Development (EATDD) requires that no code must be written for a feature unless one of the corresponding automated acceptance tests fails.

Traditional methods of requirements elicitation include interviews, questionnaires, observation and study of business documents (Maciaszek, 2001 p. 82). In Extreme Programming, requirements are gathered by creating user stories together with the customer. Then, developers create acceptance tests in collaboration with the customer who formalizes the user story into an executable and readable specification (Melnik, 2007 p. 4). When at least one feature has been translated, the development team can start to implement that feature.

However, during development it is very likely that the requirements of the system change which results in outdated acceptance tests that do not match the actual acceptance criteria. The development team uses the tests to see which features of the system are working and which still need to be implemented. Therefore, the suitable acceptance tests of the changed requirements have to be modified to be up-to-date. Once they are updated, the development team can start working on that feature.

Furthermore, acceptance tests have to be very focused on a specific feature including enough information for the development team to implement that particular feature as well as for the customer to be confident that the system works as expected. At last, acceptance tests can quickly grow in size and complexity. It is not easy to get the accurate test definition at first try.

1.2 Refactoring of Acceptance Tests

As mentioned before, by following the Executable Acceptance Test Driven Development approach the acceptance tests and the actual system requirements have to be up-to-date all the time. Whenever the requirements of the system under development change, one or more acceptance tests have to be modified.

In large-scale software development projects, the amount of acceptance tests and their size can grow very quickly which results in a large test database. Carrying out modifications in such an environment is time-consuming and error-prone. Additionally, the fixture that translates the test cases into system calls has to be kept consistent with the test definition.

Furthermore, whenever production code is refactored unit tests can be used to check whether the system's behaviour has been kept unchanged. Acceptance tests lack such an important regression safety net and thus modifications have to be made safely to minimize the risk of an unwanted behaviour change.

Acceptance test refactoring helps to modify acceptance tests in a safe and less error-prone way. Furthermore, automated acceptance test refactoring lowers the test maintenance effort in the same way as source code refactoring tools lower it for source code updates.

1.3 Thesis Goals

This thesis has two goals:

The first goal is to find applicable ways to refactor acceptance tests. Although it is possible to refactor manually, tool support is considered crucial. Therefore, the second goal is to extend the existing functional testing development environment (FTDE) FitClipse with the capability of automated refactoring of acceptance tests.

1.4 Thesis Structure

This thesis is organized as follows:

Chapter 2 presents an overview of publications that are related to the scope of this work. The following chapter 3 introduces the reader to agile software development, the agile method Extreme Programming (XP) and the testing techniques utilized. This builds a bridge to one of the core practices of XP, Test Driven Development. Chapter 4 discusses Executable Acceptance Test Driven

Development, Tools and issues of manual test modification and thus the motivation of this work. The refactoring approach developed in chapter 5 is the foundation for the following implementation of automated refactoring support in FitClipse which is described in Chapter 6. The thesis concludes with a summary and the future work.

2 Related Work

2.1 Test Refactoring in General

Test refactoring in general has been discussed by several authors. However, the publications address unit test refactoring rather than acceptance test refactoring. Although unit testing is not the focus of this work, the following will give a short overview of the work that has been done in this area.

Beck refactored production code as well as the appropriate unit test code in his examples but he only looked at it on the level of code rather than on a level of purpose (Beck, 2003). However, it showed that test code refactoring might be needed when production code is refactored but not explicitly.

Deursen et al. published a paper (Deursen, et al., 2001) where they identified 11 test code smells, which have a negative impact on the readability or maintainability. Additionally, they presented a set of 6 ways of refactoring to avoid these problems. It primarily focused on the practice and did not address issues like test code refactoring in a safe way. Nevertheless, they were the first ones who differentiated between production code and test code refactoring.

In a later publication, Deursen and Moonen divided the production code refactoring into 5 different types as mentioned by Fowler (Fowler, 2000) and they used this classification to identify which refactoring types affect the test code (Deursen, et al., 2002). With this contribution, it was possible to define which refactoring types must be applied after a production code refactoring has been performed.

Based on these results, Guerra and Fernandes (Guerra, et al., 2007) developed a graphical representation of the structure of JUnit tests to verify whether a test refactoring has been carried out without changing the behaviour. They also created a catalogue of different types of unit test code refactoring in their paper.

2.2 Effectiveness of Acceptance Tests

Andrea discussed typical problems of functional tests and presented a list of characteristics which enables functional tests to be effective requirement artefacts (Andrea, 2005). In addition, she modified an ineffective test (based on the presented characteristics) to simplify its structure and improve the readability. The five characteristics discovered are briefly described as follows:

⇒ **Declarative**

Functional tests must serve the customer and the development team. The customer must be able to see that the system satisfies the acceptance criteria. In contrast, developers must be able to read and understand the tests to know what they have to code. Declarative tests are written in the language of the business domain describing the requirements rather than in a language of a graphical user interface or the application-programming interface.

⇒ **Succinct**

The purpose of acceptance tests is to describe requirements of a software system in a comprehensive way. Large test definitions are hard to follow and not focused enough to be easy understandable. Therefore, functional tests should be kept small and to the point.

⇒ **Autonomous**

A functional test is typically read by many different readers including the customer and developers. It is important that the test is self-contained so that every reader understands it the same way. Missing preconditions should be avoided so that readers are not confused about the origin of data that is processed in the test. Furthermore, every test must be able to run without any dependencies on other tests or suites.

⇒ **Sufficient**

Functional tests should not be overloaded with all possible test cases. Rather than testing all input combinations of a system, they should describe and test business rules. While unit tests fill in the granular detail for business rules, functional tests focus on the important workflow scenarios and key business rules.

⇒ **Locatable**

Each individual functional test is just one piece of a puzzle. In order to effectively serve as a requirements specification, the reader must be able to connect the pieces together into a complete picture (Andrea p. 30). In large-scaled systems, the number of acceptance tests can grow very quickly. In this case, a database of well-named acceptance tests can help developers to find all tests related to a feature easily and makes the developing more efficient.

Not only does this show that acceptance tests can be improvable but it also shows the need to modify them. Functional tests might be ineffective and in that case need to be changed to be valuable for the development team as well as for the customer.

2.3 Requirements on Next Generation Functional Testing Tools

Andrea also discussed requirements on the next generation of functional testing tools (Andrea, 2007 p. 61). One important aspect of writing functional tests is that they have to be easy and safe to maintain. Unit tests and acceptance tests of a system build a regression safety net that helps developers at any time to ensure that the application behaves as expected. Whenever production code is refactored, this regression safety net makes changes of the behaviour or introduced bugs visible.

In contrast, functional tests do not have such a regression safety net thus the functional testing tool must support the user with powerful and safe refactoring capabilities to keep functional tests effective. They are even more important for a Functional Testing Development Environment (FTDE) than for an Integrated Development Environment (IDE).

2.4 Current Tool Support

There are several functional testing tools available that support Executable Acceptance Test Driven Development like FitNesse (FitNesse, 2008), AutAT (AutAT, 2005), ConFIT (ConFIT, 2007), FITpro (Luxoft, 2007) and GreenPepper (Pyxis, 2008). However, none of these tools supports acceptance test refactoring to carry out changes to functional tests.

3 Agile Software Development and Extreme Programming (XP)

3.1 Overview

Agile software development is a set of concepts applicable for software developing projects. This set, described by the Agile Manifesto (Manifesto, 2001), consists of four statements:

⇒ **Individuals and interactions over processes and tools**

Processes and Tools should serve only one purpose: to help the individuals involved to do their jobs better. If the processes and tools become too complex, people start having to manage the process instead of their work. A flowing communication among team members helps to keep a close relationship and increases the change for better decisions by involving the whole team.

⇒ **Working software over comprehensive documentation**

Documentation is an important artefact of software development amongst others to be able to maintain the system after it has been fully implemented. However, the goal is to produce a working software system that fits the needs of the customer. Therefore, agile teams release working systems in frequent intervals with more functionality in each step. This approach keeps the code simple and understandable and reduces the need of documentation to a minimum. Furthermore, it satisfies the customer as he can see the progress and development and is able to run the system under development at every moment.

⇒ **Customer collaboration over contract negotiation**

In agile environments, the presence of the customer is important and has advantages for both sides. The developers can clarify questions about requirements immediately and see whether they are on track or not. On the other hand, the customer stays in touch with the developing team and can step in immediately to ensure he is receiving the system he requested. Furthermore, useful ideas coming up during development process can be integrated to make the software better.

⇒ **Responding to change over following a plan**

Changing requirements are most likely in every software development project and are hard to avoid. The development should be in line with some kind of project plan but meeting the customer's goals of the project must have the highest priority. Being open for changes, having the flexibility not to follow the plan exactly and small release cycles make agile development deliver software the customer wants rather than software equivalent to a plan created at the beginning of the project.

These four statements build the conceptual base of agile methods that are described in the following.

3.2 Agile Methods

Agile Methods is a common term for a set of software development processes, which are in line with the Manifesto for Agile Software Development (Manifesto, 2001). These processes are nonlinear, iterative and lightweight and expedite the software development without compromising software quality and user satisfaction (Wang, et al., 2006 p. 308). Among others, the following methods are included: Extreme Programming (XP) by Beck (Beck, et al., 2004), Scrum by Schwaber (Schwaber, et al., 2002), Crystal Clear by Cockburn (Cockburn, 2004 p. 3), Feature Driven Development (FDD) by Palmer (Palmer, et al., 2002), Dynamic Systems Development Method by Stapleton (Stapleton, 1997) and Adaptive Software Development by Highsmith (Highsmith, 2000).

From these methods, only Extreme Programming is related to this thesis' scope of work and thus will be further discussed.

3.3 Extreme Programming (XP)

Extreme Programming (XP) is a style of software development focusing on excellent application of programming techniques, clear communication and teamwork (Beck, et al., 2004 p. 2). It was among others created by Kent Beck who realized that most common failures in software development projects can be traced to five different categories and that improvements in these areas would lead to a significantly better development process (Crispin, et al., 2002 p. 4). Extreme Programming is based on:

⇒ **Communication**

Flowing communication in the developer team including the customer improves team cohesion and helps to work together towards the same goal.

⇒ **Simplicity**

Starting with the simplest design possible and extending it systematically.

⇒ **Feedback**

Feedback is generated early and often as possible. XP strongly recommends that these feedbacks to be generated automatically.

⇒ **Courage**

Developers must be able to express their opinions with courage. This is mandatory to improve communication in the team.

⇒ **Respect**

Every member of the team needs respect from his fellows.

These five values led to a set of 12 practices that represents the rules of Extreme Programming. The primary practices are detailed described as follows:

⇒ **Sit Together as a whole team**

All members of a team should sit in a room to be able to communicate face-to-face. It helps people to work as a whole team rather than working alone. Even though they are together all time, it is important to satisfy the need of privacy by providing private spaces.

⇒ **Informative workspace**

Information about the project progress should be immediately visible by all team members. For Example, by putting story cards on a wall that shows current and upcoming tasks that give a quick overview over the state of the project.

⇒ **Pair Programming**

The idea is to let two people develop together by sitting on one machine. Pair programming is a dialog between two people simultaneously programming (and analyzing and designing and testing) and trying to program better (Beck, et al., 2004 pp. 42-43). While one person is coding, the other person reviews the code and gives immediate feedback.

⇒ **Stories**

Extreme Programming uses stories to describe units of customer-visible functionality. They include a short description of the feature and an estimation how long it will take to implement it. Developers collaborate with the customer in planning meetings to create stories and estimate the effort.

⇒ **Short iterations**

Short iterations allow the development team to react to changes and get early feedback from the customer. A small working software release is created at the end of an iteration.

⇒ **Continuous integration**

New features should be integrated into the system under development as soon as possible. The longer the team waits to integrate, the more it costs and the more unpredictable the cost becomes. In Extreme Programming, changes to the system are directly integrated and tested. The development team will be notified when an error occurs and can then start to work on the problem to solve it.

⇒ **Test first programming**

Practicing test first programming means writing a failing test first and start coding to make the test pass. Following this approach helps to increase the quality. Additionally, it helps to keep the design as simple as possible as developers have to think about what the code should do before they start working on the implementation.

In Extreme Programming, software testing is an important aspect to keep the code quality on a high level. In the following, the time and frequency of testing in Extreme Programming as well as the kinds of testing are explained.

3.4 Testing in XP

The complexity and size of today's software systems make writing of bug-free code extremely difficult, even for highly experienced programmers. The Chaos Report, created by the Standish Group, shows a staggering 31.1% of projects will be cancelled before they are completed. Further results indicate that 52.7% of projects exceeded their cost by 89% of their original estimates (The Standish Group, 1995-2005). Termination, exceeding time and budget and reduced functionality are the most common failures. The lack of sufficient testing is one of the most important reasons. Software testing is any activity aimed to evaluate an attribute or capability of a program or system determining that it meets its required results (Mathew, 2003 p. 281).

This shows the importance of testing and test first programming tries to decrease the number of bugs by following an easy approach: Write a test and make it fail before writing or changing any production code.

3.4.1 Time and Frequency of Testing

In development processes with long feedback cycles, there is a long timeframe between testing sessions where the actual codebase or system testing is performed. This means, new code that is added by developers is not tested immediately. The result is an increasing amount of defects that are very expensive to be eliminated (Beck, et al., 2004 p. 99). Additionally, many defects will remain even after testing has been performed (see Figure 3.1). Most defects end up costing more than it would have cost to prevent them (Beck, et al., 2004 p. 98). The more fixing defects costs, the more likely is it to have remaining defects in the deployed code.

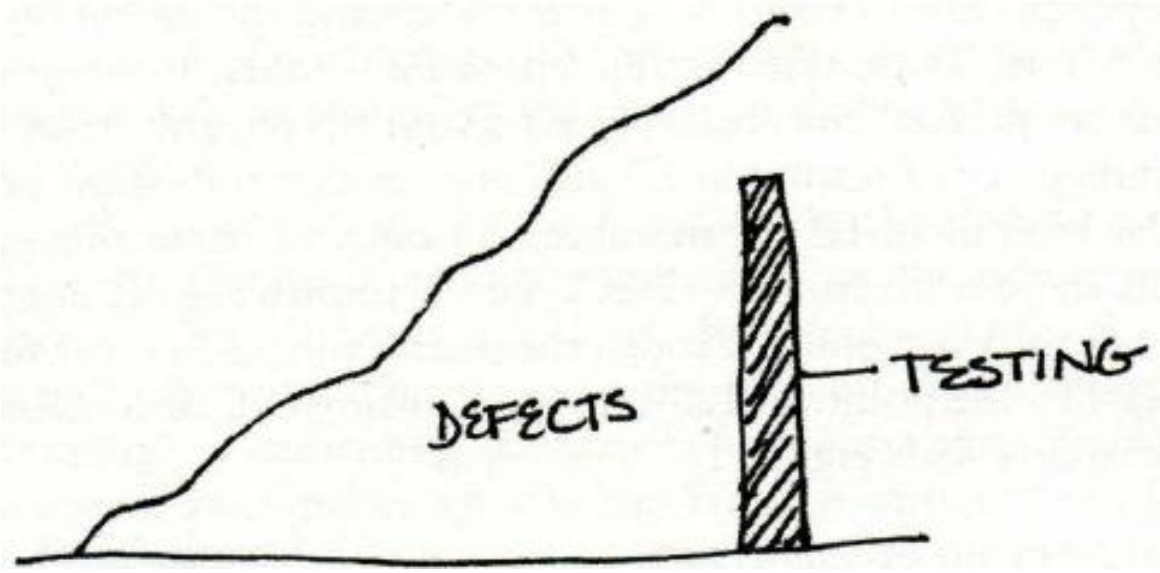

Figure 3.1: Late Testing in Development Processes with long Feedback Cycles

In contrast, Extreme Programming uses the Defect Cost Increase (DCI) principle to increase the cost-effectiveness of testing. In other words, testing is performed frequently to fix defects sooner and cheaper (see Figure 3.2). In order to prevent the number of defects to grow over the time, tests are conducted often to reduce the number of bugs.

Figure 3.2: Frequent Testing in Development Processes with short Feedback Cycles

It is much harder to remove a defect a long time after it has been introduced. The cause of the bug must be found and the affected code has to be changed in a way, so that the bug is fixed and the

rest of the program stays unchanged. To keep this effort as low as possible, Extreme Programming uses Test Driven Development that requires tests to be written before the production code.

3.4.2 Unit Testing

Unit testing represents the lowest level of testing that the system under test can undergo. Its goal is to ensure that software units meet their specified requirements. The developer, working on a software unit is responsible for designing and running a series of tests to ensure that the unit is working as specified by the requirements. Additionally to the testing aspect, unit tests serve also as documentation for developers. They show how to use the unit and the way it works in a short understandable form.

3.4.3 Acceptance Testing

Acceptance testing is a testing discipline that operates on the highest level of a software system. The purpose is to determine whether a system meets the customer's requirements (Meyers, 2004 p. 185).The term acceptance testing itself is strongly related to the agile method Extreme Programming (Beck, et al., 2004) and will be discussed in the context of this work. The following table shows most of the synonyms for which acceptance testing is known:

Table 3.1: Acceptance Testing Synonyms (Maurer, et al., 2006)

Term	Used by
Functional tests	Beck (Beck, 1999)
System tests	IEEE (IEEE, 1996), Erickson (Erickson, et al., 2003 pp. 120–128)
Formal qualification tests	US Department of Defense (US, 1988)
Soap opera tests	Buwalda (Buwalda, 2004 pp. 30-37)
Keyword-driven tests	Kaner (Kaner, et al., 2002)
Scenario tests	Kaner (Kaner, 2003)
Conditions of satisfaction	Cohn (Cohn, 2005 pp. 18-22)
Examples, business facing example, example driven development	Marick (Marick, 2003)
Coaching tests	Marick (Marick, 2002)
Specification by example	Fowler (Fowler, 2006)

Story tests and story driven development	Kerievsky (Kerievsky, 2005)
Customer tests	Beck (Beck, 1999), Jeffries (Jeffries, 2001)
Customer inspired tests	Beck (Beck, 1999)

According to the Standish Group Chaos Report (The Standish Group, 1995-2005), user involvement is the most important factor of successful projects. It is a common knowledge that more than two-thirds of all software projects today do not succeed for a variety of reasons: they are either terminated, become obsolete, exceed time restrictions or budget, or deliver a reduced set of functionality (Maurer, et al., 2006). Ambiguous and incomplete software requirements along with insufficient testing are major contributors to these failures (The Standish Group, 1995-2005). It is estimated that 85 percent of the defects in developed software originate in the requirements (Young, 2001).

The purpose of acceptance testing is to demonstrate working functionality rather than to find bugs (although bugs may be found by performing acceptance testing) (Maurer, et al., 2006). In XP, system requirements are gathered in the form of user stories. Acceptance tests should be written by the customer rather than by a developer (Cohn, 2004 p. 73). However, it has become a good practice to let a developer and the customer create test definition together. Acceptance tests have to test the system as a whole unlike unit tests, which test internal parts of the system on a very low level. Acceptance testing has the following advantages:

⇒ **Improved communication**

Acceptance testing builds a bridge between developers and customer by providing support of specifying detailed functional requirements. This helps both sides to understand the domain problem and the application better. Acceptance tests can help structure conversation within the team as well as discussions with the customer by defining a common language.

⇒ **Regression safety net**

Functional tests can serve as regression tests, which ensure that previously working functionality continues to behave as expected (Maurer, et al., 2006 p. 1). In Extreme Programming, the development team is working on the system until every acceptance test passes. Due to the fact that this process is iterative, the code base is steadily increasing and new functions are added which can cause already implemented functions to fail. In this case, acceptance tests can help to find bugs occurring in already implemented parts of the system. In other words, the development team is able to find regression failures by keeping track of passing and failing tests.

⇒ **System documentation**

Acceptance tests can also be seen as a partial replacement for documentation - especially for requirements documents (Aarniala, 2006). Following the Extreme Programming process in the long term, acceptance tests keep in sync with the actual system while requirements documents may easily lag behind. Additionally, the tests show the functionality of the system in a short and easily understandable format. This helps new developers start working on the application and makes maintenance easier.

⇒ **Development progress tracking**

Acceptance tests are an absolute criterion to decide whether a feature is complete or not. Failing tests show that the story is not implemented in a way the customer will accept it. Additionally, the amount of passing or failing tests shows developers and customer how the project development is progressing.

⇒ **Improved effort estimation**

Due to clear communication of all requirements of the system, effort estimates can be much more accurate. Developers do not have to expect hidden customer expectations suddenly popping up during development.

In Extreme Programming, test-driven development is a core practice that follows the “test first” rule. The next chapter describes TDD as well as the two techniques utilized in detail.

4 Test Driven Development as one Core Practice of XP

4.1 Unit Test Driven Development

Unit Test Driven Development (UTDD, also known as Test Driven Development or TDD) is a style of development following the test first approach. Rather than writing production code first and testing the code afterwards, developers create tests first and implement the code to make the tests pass. This implies that every piece of code is covered by tests to ensure the correctness of the system components. Figure 4.1 describes the workflow of Test Driven Development as a state diagram. It makes clear that developing of new code starts with adding the proper test. The implementation of a feature can only be seen as finished when all tests pass.

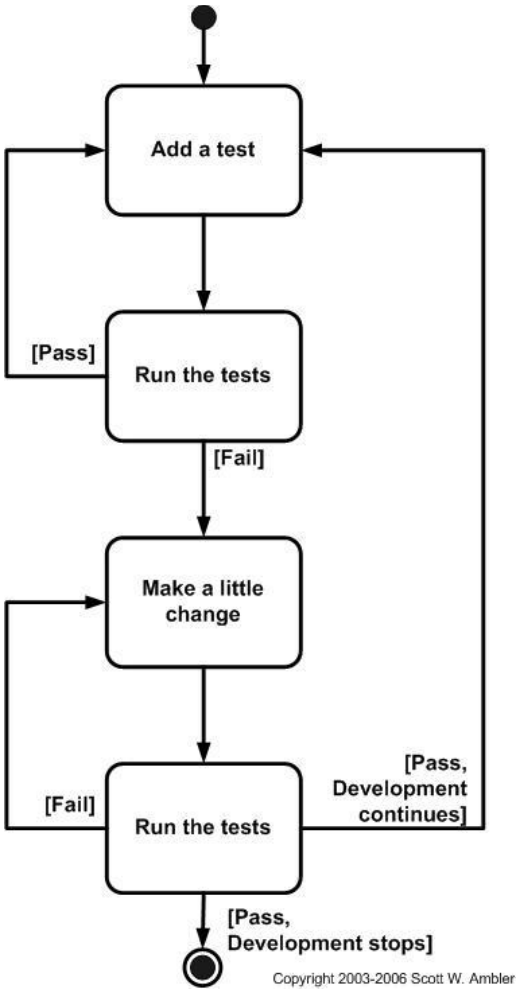

Figure 4.1: The Steps of Test First Design (Ambler, 2007)

The steps of test first design are:

⇒ **Quickly add a test to the suite**

Based on the imagination of how the application would work, invent the interface.

⇒ **Run all the tests**

Run all the tests to see the newly added test is failing. At this time, a red bar should appear.

⇒ **Make the test pass**

Change the system code to make the newly added test pass, also run the whole suite of the tests to make sure all tests pass in order to prevent breaking other parts of the system. If some other tests fail, they have to be fixed in order for the implementation to be completed.

⇒ **Improve the design**

Refactoring needs to be done to remove duplicates that have been introduced to the system and to improve the design.

This process will be repeated for every new unit (e.g. method) that is added. Once the system is finished and all tests are passing, the system might need to be extended or changed to accommodate the new requirements or to fix bugs.

Tests are written to define what it means for the code to work (Astels, 2003 p. 7). Running the tests automatically several times a day ensures that bugs introduced when adding new code can be detected immediately. Additionally, unit tests can be used to check whether all components of a software system still work as expected. This is helpful when components are refactored to see whether the behaviour has changed or not.

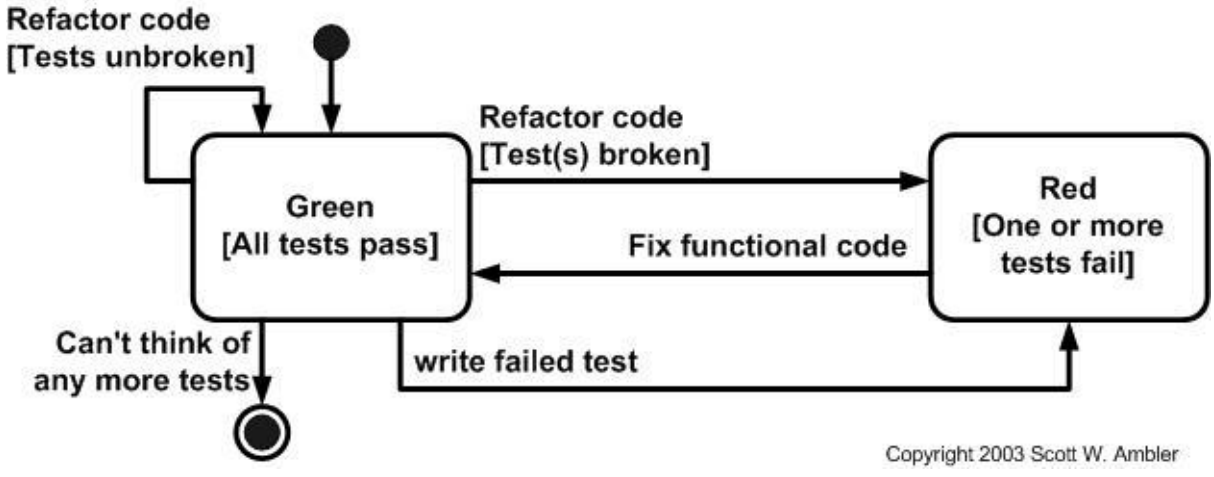

Figure 4.2: Refactoring System in Test Driven Development (Ambler, 2007)

When the system has to be changed, the production code can be refactored while the unit tests serve as a regression safety net. As long as they are passing, the developers can be sure that the system works like before. If the tests run before refactoring and they run after, you can be confident that the code behaviour has not changed (Astels, 2003 p. 493).

In contrast, when the system has to be extended, a new failing test is added. The developers extend the production code to make the new test as well as all other tests pass (see Figure 4.2).

4.2 Executable Acceptance Test Driven Development

4.2.1 Overview

Executable Acceptance Test Driven Development (EATDD) or Story Test Driven Development is similar to unit test driven development but involves writing one or more executable system-level acceptance tests for a feature before the solution. Figure 4.3 shows exemplary the output of an executed acceptance test. The purpose of acceptance tests, their structure and tools for executing acceptance tests will be discussed further in this chapter.

fitlibrary.DoFixture			
start	tests.CreateBankAccountBranch		
enter first name of applicant	Heiko	and second name	Ordelt
choose account type	chequing		
check that bank account does not exist			
create bank account			
check that bank account does exist			
check that account type is	chequing		

Figure 4.3: Example Acceptance Test Execution Output

In Extreme Programming, all acceptance tests must be automated (Crispin, et al., 2002 p. 133). Due to the iterative nature of processes in the agile world, manual regression testing at the customer level is too time consuming to be practical and feasible given the short timeframes of agile iterations (Maurer, et al., 2007 p. 245). Furthermore, manual testing has the following disadvantages:

⇒ **Unreliable**

The effectiveness of manual testing is highly dependent on schedule pressure. Whenever, the delivery date of the system under development comes closer, people start to cut corners, omit tests and miss problems (Crispin, et al., 2002 p. 134).

⇒ **Undermining the Extreme Programming testing practice**

Manual testing can attract developers to omit writing unit tests because they want to spend the saved time in implementing new features. In this case, rather than creating the appropriate unit tests, manual testing is performed and one of the most important practice of XP is undermined.

⇒ **Manual tests are divisive**

Manual testing relies on the people performing the testing. It is possible that testers fail to see something. When the stakes are high and something is missed, the blame will be fallen the testing person (Crispin, et al., 2002 p. 135). If automated tests are used, the developers could have caught the defect before it is checked by the testers.

Executable Acceptance Test Driven Development also pushes the test driven development paradigm of agile methods up to the customer level. EATDD extends this by requiring that no code is written for a new feature unless an automated acceptance test fails (ASE, 2008). The cycle of acceptance testing in Extreme Programming is shown in Figure 4.4.

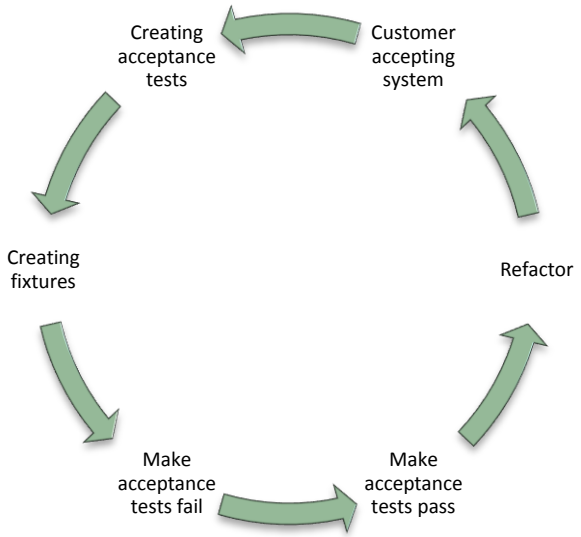

Figure 4.4: Acceptance Testing Cycle in Extreme Programming

In the following, every step is described briefly:

⇒ **Creating acceptance tests**

Based on the stories that describe the system requirements in Extreme Programming, the appropriate acceptance tests are created in collaboration with the customer and the developers.

⇒ **Creating fixtures**

To make the created acceptance tests executable against the system under development the developers create fixtures for all defined acceptance tests.

⇒ **Make the acceptance tests fail**

As the feature described by the acceptance test has not been implemented yet the acceptance tests must fail. This helps the team to see which features are finished and which requirements are left to implement.

⇒ **Make the acceptance tests pass**

Once the development has started, the developers have to follow the Unit Test Driven Development approach to create the production code needed to make the acceptance tests pass, one by one.

⇒ **Refactor**

After the tests pass, the written code has to be refactored to delete duplicates, improve the design and make the code better understandable.

⇒ **Customer accepting system**

When all acceptance tests pass, the customer can run them to see whether the system works as it is expected. If the customer accepts the system, the development can be seen as completed. If not, the appropriate tests have to be modified and the process starts again.

As shown in this chapter, all acceptance tests must be automatically executable as manual acceptance testing is not feasible and has many disadvantages.

4.2.2 Tools

4.2.2.1 Fit Framework and FitLibrary

The Framework for Integrated Tests (FIT) (Framework) is used for executable acceptance testing. It is probably the most popular functional testing framework today. Tests written for FIT consist of two parts: test definition and fixture (see Figure 4.5).

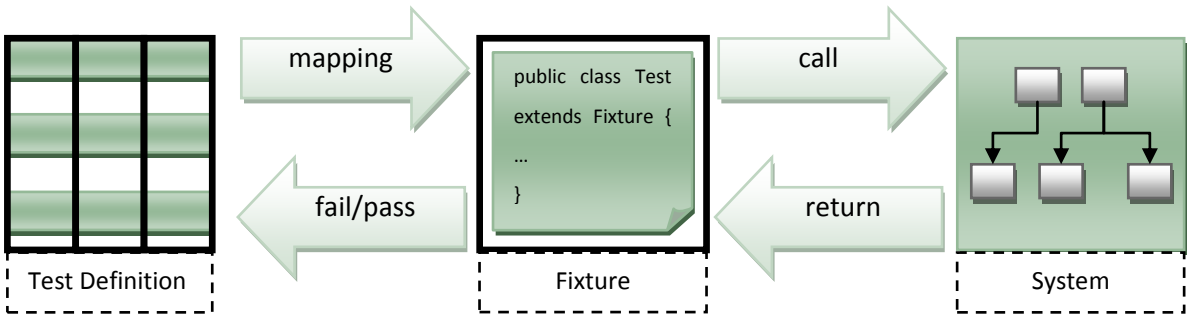

Figure 4.5: Executable Acceptance Testing Workflow

The test definition can be created with business tools like word processors. Developers are responsible for writing the fixture that is in charge of making the appropriate method calls in the system under test. The fixture is written by the developers in the programming language of the system that is supposed to be tested. FIT maps then the test cases in the test definition to the fixture and returns the result of the test run. These results are shown by using three different color codes:

⇒ **Green**

The test case ran successfully. The expected result specified in the test case equals the returned value of the system.

⇒ **Yellow**

An exception occurred. For example, the fixture is not consistent with the test definition or an exception is thrown by the system under test.

⇒ **Red**

The test case ran successfully but the returned result did not match the expected result.

FIT supports different formats of tests depending on the purpose of the test. One format, for example, is testing of workflows or computations. An extended set of fixtures is provided by the FitLibrary (FitLibrary) to allow FIT to run tests suitable for even more purposes.

4.2.2.2 FitNesse

FitNesse is a Wiki front-end testing tool which supports team collaboration to create and edit acceptance tests. FitNesse uses the Fit framework to run acceptance tests via a web browser. It also integrates FitLibrary fixtures for writing and running acceptance tests (Deng, et al., 2007). FitClipse uses the wiki syntax introduced by FitNesse to define acceptance tests.

4.2.3 Multi-Modal Test Execution

Software systems are often built in multiple layers (e.g. persistence layer, business layer and user interface) (Gamma, 1995). This multi-layered architecture leads to better maintainable and extendable systems. For example, modifications to one layer might only have an impact on the layer directly above which needs less effort when changing a part of the system. When using this architecture style, a new functionality is implemented across many layers. For example, the business layer could be responsible for calculating and maintaining business rules, while the user interface is responsible for gathering the input data and displaying the output result. These layers have to be properly integrated in order for the functionality to become useful.

In multi-layered systems, a feature appears in different layers of the software architecture or different components of the software. While following the Executable Acceptance Test Driven Development approach, this feature must be covered by an acceptance test. This leads to duplication of the test for all different layers that is time-consuming and error-prone to maintain.

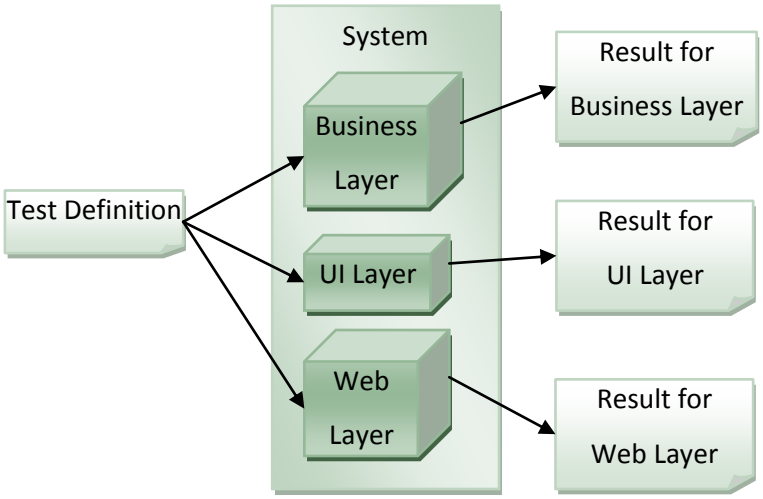

Figure 4.6: Multi-modal test execution

Multi-modal test execution provides a one-to-many mapping between test definition and fixtures. In other words, a test is executed against different layers or components of a software system by calling different fixtures (Park, et al., 2008) (see Figure 4.6).

4.2.4 Manual Acceptance Test Modification Issues

As shown in chapter 2, acceptance tests might have to be modified to improve their effectiveness und to make them better understandable. Additionally, a requirement may change, be removed, or

a new requirement may be added at any phase of the development lifecycle (Maciaszek, 2001 p. 92). During software development lifecycle, a change of the customer's requirements is very likely and results in changed acceptance criteria. Due to the direct relationship to one or more acceptance tests, these tests are outdated and are supposed to be adjusted as well.

Figure 4.7 shows the effect of changing requirements in Executable Acceptance Test Driven Development. In this example, requirement A is split into two features covered by acceptance test A and B. Acceptance test A is linked to fixture A.1 and A.2 while acceptance test B is linked to fixture B. Obviously, every fixture makes calls in the system. The lines in red shows the affected elements of the overall test structure when requirement A has changed. This clarifies that a small requirement change can have a huge impact on several testing elements.

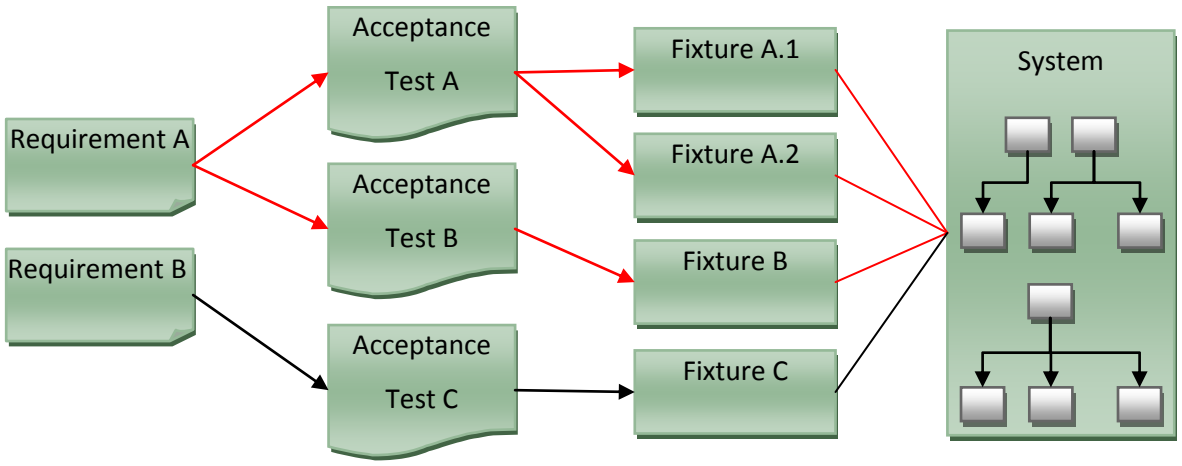

Figure 4.7: Effect of changing Requirements in Executable Acceptance Test Driven Development

Therefore, a manual adjustment of existing acceptance tests has the following disadvantages:

⇒ **Time-consuming**

Big projects can have hundreds of features and therefore a large set of acceptance tests can exist. As shown in Figure 4.7, a requirement can be associated with several acceptance tests that are linked to multiple fixtures. Making changes in such test environments not only consists of making the appropriate modifications but also finding relationships between requirements, acceptance tests and fixtures. This makes carrying out changes very time-consuming.

⇒ **Error-prone**

Even though acceptance tests should be kept small to make them better understandable, this cannot always be achieved which makes it even harder to change the tests manually. When acceptance tests are modified, the appropriate fixtures have to be kept consistent (see Figure 4.7). This can easily lead to errors in test definition or execution failures due to fixtures that have not been updated. Therefore, every change has to be applied very carefully to minimize the risk of unexpected failures.

⇒ **Regression safety net**

Acceptance tests along with unit tests are used to check whether a software system works the way it is supposed to. When the system code base is changed, e.g. refactoring the source code to improve the design, regression testing can be used to verify if the behaviour has changed. In contrast, when acceptance tests are changed there is no guarantee that the test behaviour is the same.

To provide a way of safe and consistent modifications of acceptance tests, the next chapter introduces the developed refactoring approach. It builds the bridge for the following implementation of automated refactoring support.

5 Refactoring of Acceptance Tests

5.1 Goal of Acceptance Test Refactoring

As shown in 4.2.4 and discussed by Andrea (Andrea, 2005), acceptance tests might be modified to improve their readability and to follow requirement changes. It was also shown that the manual modification of acceptance tests is time-consuming, error-prone and might lead to an unexpected change of the behaviour. Therefore, changes to acceptance tests have to be applied safely to avoid those problems. Additionally, the fixture that translates the test cases into system calls has to be kept consistent with the test definition. As a conclusion, the goal of acceptance test refactoring is a consistent modification of test definition and fixture.

Refactoring of source code is already well known and is typically used to improve the design of code and make it better to understand. Since source code refactoring is very similar to the goal of acceptance test refactoring, it will be discussed further and the differences will be shown.

5.2 Source Code Refactoring

Source code refactoring is a technique to change source code with one main constraint: The behaviour of the code must be preserved. It is defined as “a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour” (Fowler, 2000 p. 53).

In general, refactoring is not intended to fix bugs or add new functionality although bugs can be discovered (Fowler, 2000 p. 57). Refactoring has three main purposes of use:

⇒ **Improving the design of software**

Over time, the design of programs decays. As developers who have to add small features tend to make changes without a full comprehension of the design the code loses its structure. Among others, bad code design leads to duplicated code and makes future modifications more expensive as developers have to understand more code.

⇒ **Making software easier to understand**

Developers are typically focusing to get the code working rather than making it easy to understand. Other developers who work on the same code later on have to spend much more time to understand the code and to make their changes. This situation can be avoided by spending a little time after adding new functionality for code cleaning and design improvement. In case the rule was not executed, to refactor code to make it comprehensible refactoring even helps developers to understand code they have not seen before.

⇒ **Finding bugs**

Trying to understand the code also helps to spot bugs. Refactoring means working on the understanding what the code does and putting that understanding back into the code. In that process a clarification takes place and bugs will be found.

All the points mentioned above can help to develop code more quickly (Fowler, 2000 p. 57). Furthermore, refactoring is used to remove bad code smells (Fowler, 2000 pp. 75-87) which typically makes code complicated and hard to maintain.

In Test Driven Development, refactoring is integrated in the development cycle. It is performed by the developers after they finished working on the system to make the unit tests pass, to improve the code's internal consistency and clarity. To ensure that the behaviour of refactored code has not changed unit testing can be used. This means, the unit tests run before and after the refactoring. If both results are the same, the change has been carried out without changing the behaviour.

5.3 Definition and Separation from Source Code Refactoring

Fowler's (Fowler, 2000 p. xvi) definition of refactoring explains the motivation for acceptance test refactoring just partially. Some changes to acceptance tests are dealing with making them more readable and more understandable and would fit under Fowler's definition.

However, other changes in acceptance tests will change the external behaviour, as the intention is to change the specification, e.g. removing a column of a test. In this case, the goal is to allow the user to carry out those changes safely and to keep the test definition and the corresponding fixture consistent.

Additionally, depending on the refactoring it cannot be ensured that the external behaviour stays the same. While analyzing acceptance test refactoring, two different kinds of refactoring could be identified based on the state of the behaviour after applying the modifications:

⇒ **Behaviour preserving**

The behaviour specified by the acceptance test has not been changed by the refactoring and there is no user interaction needed to make the refactored test pass.

⇒ **Behaviour changing**

The behaviour specified by the acceptance test has been changed by the refactoring and the user is required to update the fixture and/or system under test manually to make the refactored test pass.

This classification is important as not all kinds of acceptance test refactoring are behaviour preserving (see Table 5.1 and Table 5.2 for details). However, the refactoring has to result in a successful compilation of the fixture code and in executable acceptance test. Acceptance test refactoring is defined as:

Acceptance test refactoring is the process of changing an acceptance test definition and the corresponding fixture class so that the fixture class compiles successfully and test execution results in either

⇒ ***success (green)***

whenever the change does not change the external behaviour of the system (behaviour preserving) and

⇒ ***fail (red)***

whenever the change does change the external behaviour of the system (behaviour changing).

In other words, changes to an acceptance test definition and/or its corresponding fixture should never result in exceptions being thrown due to a mismatch between the test definition and the fixture code.

Acceptance test refactoring tools can lower the test maintenance effort in the same way as code-refactoring tools lower it for updates of the source code. Furthermore, automated refactoring tool support can reduce the risk of inconsistencies between the test definition and the fixture code when acceptance tests are changed.

5.4 Analysis of Fit Based Acceptance Tests

To develop an acceptance test refactoring catalogue, the relationships between test definitions and fixtures based on the FIT framework (Cunningham) and the FitLibrary extension (FitLibrary) have been analyzed.

Although the FIT framework supports many different formats to specify the test definition, only wiki code introduced by FitNesse (FitNesse, 2008) was used for the following analysis as well as the implemented refactoring support.

Additionally, the work was initially focused on ColumnFixture and DoFixture, as these seem to be most widely used fixture types (based on informal discussions with industry contacts).

FIT tests are written as tables in HTML and based on the used fixture the cells have different meaning. ColumnFixture is often used to test computations. Test cases are written as tables with given and expected-value columns. Given-value columns represent input parameters and expected-value columns contain the anticipated output.

The first acceptance test example (see Figure 5.1) shows a test table that can be processed by a ColumnFixture in the upper part and the corresponding fixture code in the lower part. It tests whether a person's credit application will be approved or not. Various criteria for consideration of the applicant are if he is a staff member, the actual balance and how long the person has been a customer of the bank. As output of the system, the approval decision of whether the credit application is shown and, if yes, the credit limit is checked.

A.1	tests.CreditCondition				
A.2	staff	balance	months	allow credit()	credit limit()
A.3	false	6000	12	true	3000
A.4	true	3000	12	true	5000

```
F.1     public class CreditCondition extends ColumnFixture {
F.2             public double balance;
F.3             public int months;
F.4             public boolean staff;
F.5
F.6             public boolean allowCredit() {...}
F.7
F.8             public double creditLimit() {...}
F.9
F.10            private double creditLimitCustomer() {...}
F.11
F.12            private double creditLimitStaff() {...}
F.13    }
```

Figure 5.1: Example of Test Definition and Fixture (ColumnFixture)

The following relationships can be detected:

⇒ The fixture class linked with this test is referenced in A.1

⇒ The given-value columns *staff*, *balance* and *months* in A.2 are mapped to corresponding fields in the fixture class (see F.2, F.3 and F.4).

⇒ The expected-value columns *allow credit()* and *credit limit()* are mapped to the methods *allowCredit()* and *creditLimit()* in the fixture code (see line F.6 and F.8).

In general, given-value columns are mapped to fields and expected-value columns are mapped to methods. The FIT framework supports all primitive and non-primitive Java data types by automatically converting the input to the field's data type in the fixture.

DoFixture describes a sequence of actions in a workflow. An action is defined in a single row. A row starts with a keyword in the first cell. Every odd cell of the row also contains keywords. Every keyword has an associated value.

A.1	fitlibrary.DoFixture			
A.2	start	tests.CreateBankAccountBranch		
A.3	enter first name of applicant	Heiko	and second name	Ordelt
A.4	choose account type	chequing		
A.5	check that bank account does not exist			
A.6	create bank account			
A.7	check that bank account does exist			
A.8	check that account type is	chequing		

```
F.1   public class CreateBankAccountBranch extends DoFixture {
F.2
F.3           public boolean enterFirstNameOfApplicantAndSecondName(String
      applicant, String name) {...}
F.4
F.5           public boolean chooseAccountType(String type) {...}
F.6
F.7           public boolean createBankAccount() {...}
F.8
F.9           public boolean checkThatBankAccountDoesExist() {...}
F.10
F.11          public boolean checkThatBankAccountDoesNotExist() {...}
F.12
F.13          public boolean checkThatAccountTypeIs(String is) {...}
F.14  }
```

Figure 5.2: Example of Test Definition and Fixture (DoFixture)

The second acceptance test example (see Figure 5.2) shows a test table that can be processed by a DoFixture in the upper part and the corresponding fixture code in the lower part. It tests the process of creating a chequing bank account as a workflow. First, the data of the account owner is entered and the proper account type is chosen. The next step is to create the account and afterwards verify if

the account has been created and that it has the proper type. The following relationships can be detected:

- ⇒ The fixture class linked with this test is referenced in A.2
- ⇒ The keywords of the actions (see lines A.3 to A.8) are camel-cased and mapped to methods in the fixture class in line F.3 to F.13.
- ⇒ The parameters of the actions are mapped to parameters of the corresponding methods in order of appearance.

In general, actions are linked to methods in the fixture. In contrast to ColumnFixture, there are no fields used in DoFixture.

Multiple Fixtures

The ASE group of the University of Calgary has recently extended the FIT framework as well as the FitLibrary to support multi-modal test execution as shown in 4.2.3. Since this extension is not officially supported, multiple fixtures were not considered in the performed analysis to allow people to reproduce the presented results. However, an “Rename Acceptance Test” refactoring example with multiple fixtures is given in 5.5.1 which shows the syntax and how the refactoring could be carried out for an easier understanding. In addition, the developed refactoring extension for FitClipse supports multiple fixtures for all refactoring types. The appropriate implementation details are explained in chapter 6.6.

5.5 Refactoring Catalogue

Based on the results of the analysis of the FIT framework the following catalogue of refactoring types has been created. It summarizes each kind of refactoring briefly.

A refactoring is described by the type of fixture it can be applied to, a name, the input it expects, special conditions which might have an impact on the changes, the changes to the test definition as well as fixture and whether it changes the test behaviour or not. The alteration to the test definition or fixture can be different at any time, based on the input or special conditions of the fixture.

The following catalogue describes all refactoring types for ColumnFixture:

Table 5.1: Refactoring Catalogue for ColumnFixture

Refactoring	Input	Special Conditions	Changes Test Definition	Changes Fixture	Behaviour Changing
Rename acceptance test	New name	None	Rename referenced fixture in cell [0,0]	Rename class name Rename constructor if present	No
Add column	Given-value column name, Position	None	Add column at chosen position	Add field of type String and chosen name	No
Add column	Expected-value column name, Position	None	Add column at chosen position	Add method with chosen name with return value String and no parameters Add false return	Yes
Remove column	Given-value column	Field not referenced	Remove column	Remove field of column	No
Remove column	Given-value column	Field referenced	Remove column	Remove field of column Comment out the body of all methods that reference the removed field and all methods that call commented methods, change return value to String, add a reminder message as return value and a TODO comment	Yes
Remove column	Expected-value column	Method not referenced	Remove column	Remove method	No
Remove column	Expected-value column	Method referenced	Remove column	Remove method of column Comment out the body of all methods that reference the removed method and all methods that call commented methods, change return value to String, add a reminder message as return value and a TODO comment	Yes

The following catalogue describes all refactoring types for DoFixture:

Table 5.2: Refactoring Catalogue for DoFixture

Refactoring	Input	Special Conditions	Changes Test Definition	Changes Fixture	Behaviour Changing
Rename acceptance test	New name	None	Rename referenced fixture in cell [1,1]	Rename class name Rename constructor if present	No
Rename action	New name of existing action	None	Rename all occurrences of action	Rename methods corresponding to the changed action Rename method calls to method of changed action	No
Add action	New action, position	None	Add action at specified position	Add method named by camel-casing keywords with corresponding parameter Add false return	Yes
Remove action	Action, position	Method not referenced No occurrence of removed action left	Remove action at specified position	Remove method of action	No
Remove action	Action, position	Method not referenced At least one occurrence of removed action left	Remove action at specified position	None	No
Remove action	Action, position	Method referenced No occurrence of removed action left	Remove action at specified position	Remove method of action Comment out the body of all methods calling the removed action method, change return value to String, add a reminder message as return value and a TODO comment	Yes

Refactoring	Input	Special Conditions	Changes Test Definition	Changes Fixture	Behaviour Changing
Remove action	Action, position	Method referenced At least one occurrence of removed action left in test definition	Remove action at specified position	None	Yes

Next, each refactoring of the catalogue will be described in more detail. The following description structure has been used:

- ⇒ Name
- ⇒ Motivation
- ⇒ Summary
- ⇒ Mechanics
- ⇒ Examples

The name of the refactoring is unique and is used in this work to refer this one particular refactoring. After the name, a motivation is given which shows the reasons for that refactoring from a non-technical or business facing view. The summary shows the situations in which the refactoring can be applied. In the third step, the mechanics explain systematically how to apply the refactoring to the test definition as well as the fixture. This can differ between refactoring tasks, based on the input or the existing structure of the fixture. As shown before, each refactoring is either behaviour changing or behaviour preserving. Lastly, at least one example is given that shows in detail how the particular refactoring works. In some cases, this will lead to different examples showing different behaviours.

5.5.1 Rename Acceptance Test

Motivation

The acceptance tests together with the fixtures of a project are stored in a repository so that every team member can easily access and modify them. Additionally, a shared test repository is needed to ensure that developers are working on the latest and most up-to-date tests. To distinguish between acceptance tests and the system features which are supposed to be tested, each acceptance test has a name associated. In addition, every pair of acceptance tests and fixtures is named equally. For example, the test *CreateBankAccountBranch* could be associated with the fixture

CreateBankAccountBranch.java. This structures the test database in an easy manageable way and helps developers who are working on an acceptance test to find the corresponding fixtures quickly.

Summary

Whenever an acceptance test is supposed to be renamed, the corresponding fixture also has to be renamed. In this case, the "Rename Acceptance Test" refactoring can be used to easily change the name and ensure that the linked fixture is updated automatically.

Mechanics

The "Rename Acceptance Test" refactoring is carried out in the following way:

1. Rename acceptance test.
2. Change the name of linked fixture in the test definition in the corresponding cell
    a. in ColumnFixture cell [0,0].
    b. in DoFixture cell [1,1].
3. Change the filename of the fixture to new name.
4. If present, change name of constructor in fixture to new name.

How the name of the acceptance test is saved can differ with respect to the storage system that is used to save the test database. This work assumes that the storage system is able to rename an acceptance test and does not give any details on the actual operation.

Example: Single Fixture

In the simplest case, the acceptance test is linked to exactly one fixture. The following example shows an acceptance test definition and the linked fixtures (DoFixture) before and after the "Rename Acceptance Test" refactoring. The test is saved in the storage system as *CreateBankAccountBranch* and is associated with the fixture *CreateBankAccountBranch* (see Figure 5.3).

A.1	fitlibrary.DoFixture			
A.2	Start	tests.CreateBankAccountBranch		
A.3	enter first name of applicant	Heiko	and second name	Ordelt
A.4	choose account type	chequing		
A.5	check that bank account does not exist			
A.6	create bank account			
A.7	check that bank account does exist			
A.8	check that account type is	chequing		

F.1	`public class CreateBankAccountBranch extends DoFixture {`
F.2	`...`
F.3	`}`

Figure 5.3: Test Definition and Fixture before "Rename Acceptance Test" Refactoring with one Fixture

The "Rename Acceptance Test" refactoring is applied to rename the test from *CreateBankAccountBranch* to *CreateBankingAccountBranch*. The Figure 5.4 shows the changed test definition and fixture (the changes are highlighted in bold).

A.1	fitlibrary.DoFixture			
A.2	start	**tests.CreateBankingAccountBranch**		
A.3	enter first name of applicant	Heiko	and second name	Ordelt
A.4	choose account type	chequing		
A.5	check that bank account does not exist			
A.6	create bank account			
A.7	check that bank account does exist			
A.8	check that account type is	chequing		
F.1	`public class` **`CreateBankingAccountBranch`** `extends DoFixture {`			
F.2	`...`			
F.3	`}`			

Figure 5.4: Test Definition and Fixture after "Rename Acceptance Test" Refactoring with one Fixture

The acceptance test has been renamed to *CreateBankingAccountBranch* in the storage system, the linked fixture in the second cell in A.2 has been updated and the fixture class in F.1 has been renamed, too.

Example: Multiple Fixtures

A more sophisticated case is an acceptance test (DoFixture) linked to multiple fixtures. The following example (see Figure 5.5) shows a test that is saved in the storage system as *CreateBankAccount* and is associated with the fixtures *CreateBankAccountBranch* and *CreateBankAccountWeb*.

A.1	fitlibrary.DoFixture			
A.2	start	tests.CreateBankAccountBranch, tests.CreateBankAccountWeb		
A.3	enter first name of applicant	Heiko	and second name	Ordelt
A.4	choose account type	chequing		
A.5	check that bank account does not exist			
A.6	create bank account			
A.7	check that bank account does exist			
A.8	check that account type is	chequing		
F.1	`public class CreateBankAccountBranch extends DoFixture {`			
F.2	`...`			
F.3	`}`			
F.4				
F.5	`public class CreateBankAccountWeb extends DoFixture {`			
F.6	`...`			
F.7	`}`			

Figure 5.5: Test Definition and Fixtures before "Rename Acceptance Test" Refactoring with Multiple Fixtures

The refactoring is applied to rename the test from *CreateBankAccount* to *CreateBankingAccount*. It results in the following test definition and fixtures:

A.1	fitlibrary.DoFixture			
A.2	Start	**tests.CreateBankingAccountBranch, tests.CreateBankingAccountWeb**		
A.3	enter first name of applicant	Heiko	and second name	Ordelt
A.4	choose account type	chequing		
A.5	check that bank account does not exist			
A.6	create bank account			
A.7	check that bank account does exist			
A.8	check that account type is	chequing		

```
F.1  public class CreateBankingAccountBranch extends DoFixture {
F.2  ...
F.3  }
F.4
F.5  public class CreateBankingAccountWeb extends DoFixture {
F.6  ...
F.7  }
```

Figure 5.6: Test Definition and Fixtures after "Rename Acceptance Test" Refactoring with Multiple Fixtures

The acceptance test has been renamed to *CreateBankingAccount* in the storage system, the linked fixtures in the second cell in A.2 have been updated and fixture classes in F.1 and F.5 have been renamed.

5.5.2 ColumnFixture

In the refactoring descriptions for ColumnFixture the following nomenclature is used:

⇒ Column names ending with "()" are expected-value columns and are associated with a public method without parameters which name equals the camel-cased column name.

⇒ Column names not ending with "()" are given-value columns and are associated with a public field in the fixture which name equals the camel-cased column name.

5.5.2.1 Add Column

Motivation

Column based acceptance tests as ColumnFixture are mostly used to test computations. They take various input parameters and compare the output to the expected result. For example, a system that is calculating the credit limit of a credit application could have as input parameters if applicant is a staff member, the actual account balance and the duration of the membership. During development, this feature could change to incorporate more parameters, like the information whether the applicant already has a credit card, which has to be included in the decision.

Summary

Whenever requirements change and a column based acceptance test has to be adjusted to include more given-value or expected-value columns, the “Add Column” refactoring can be used to add the columns needed.

Mechanics

The “Add Column” refactoring is carried out in the following way:

1. Add new column at the specified position in the test definition.
2. Add value “TODO” in every cell of the new column beginning after the table captions row in the test definition.
3. If the new column is an expected-value column, add a new corresponding method with return type String to the fixture.
4. If the new column is a given-value column, add a new corresponding field of type String to the fixture.

Example: Add Given-Value Column

The following example shows an acceptance test definition and the linked fixtures (ColumnFixture) before and after the “Add Column” refactoring. The test is saved in the storage system as *CreditCondition* and is associated with the fixture *CreditCondition* (see Figure 5.7).

A.1	tests.CreditCondition				
A.2	staff	balance	Months	allow credit()	credit limit()
A.3	false	6000	12	true	3000
A.4	true	3000	12	true	5000

```
F.1   public class CreditCondition extends ColumnFixture {
F.2           public double balance;
F.3           public int months;
F.4           public boolean staff;
F.5
F.6           public boolean allowCredit() {...}
F.7
F.8           public double creditLimit() {...}
F.9
F.10          private double creditLimitCustomer() {...}
F.11
F.12          private double creditLimitStaff() {...}
F.13  }
```

Figure 5.7: Test Definition and Fixture before “Add Column” Refactoring of Given Value Column

The “Add Column” refactoring is applied with *credit card* as new column and “after months” as specified position.

A.1	tests.CreditCondition					
A.2	staff	balance	months	**credit card**	allow credit()	credit limit()
A.3	false	6000	12	**TODO**	true	3000
A.4	true	3000	12	**TODO**	true	5000

```
F.1   public class CreditCondition extends ColumnFixture {
F.2           public String creditCard;
F.3           public double balance;
F.4           public int months;
F.5           public boolean staff;
F.6
F.7           public boolean allowCredit() {...}
F.8
F.9           public double creditLimit() {...}
F.10
F.11          private double creditLimitCustomer() {...}
F.12
F.13          private double creditLimitStaff() {...}
F.14  }
```

Figure 5.8: Test Definition and Fixture after “Add Column” Refactoring of Given Value Column

The Figure 5.8 shows the changed test definition and fixture (the changes are highlighted in bold). The test definition has a new column *credit card* in A.2 and the fixture has a new public field *creditCard* of type *String* in F.2.

Example: Add Expected-Value Column

Besides the given-value column, the “Add Column” refactoring can also be used to add expected-value columns. The following example uses the same test definition and fixture as before and shows the effect of adding an expected-value column with the “Add Column” refactoring.

A.1	tests.CreditCondition				
A.2	staff	balance	Months	allow credit()	credit limit()
A.3	false	6000	12	true	3000
A.4	true	3000	12	true	5000

```
F.1   public class CreditCondition extends ColumnFixture {
F.2           public double balance;
F.3           public int months;
F.4           public boolean staff;
F.5
F.6           public boolean allowCredit() {...}
F.7
F.8           public double creditLimit() {...}
F.9
F.10          private double creditLimitCustomer() {...}
F.11
F.12          private double creditLimitStaff() {...}
F.13  }
```

Figure 5.9: Test Definition and Fixture before “Add Column” Refactoring of Expected-Value Column

The “Add Column” refactoring is applied with *credit card()* as new column and “after credit limit()” as specified position.

A.1	tests.CreditCondition					
A.2	staff	balance	months	allow credit()	credit limit()	**credit card()**
A.3	false	6000	12	true	3000	**TODO**
A.4	true	3000	12	true	5000	**TODO**

```
F.1   public class CreditCondition extends ColumnFixture {
F.2           public double balance;
F.3           public int months;
F.4           public boolean staff;
F.5
F.6           public boolean allowCredit() {...}
F.7
F.8           public double creditLimit() {...}
F.9
F.10          private double creditLimitCustomer() {...}
F.11
F.12          private double creditLimitStaff() {...}
F.13
F.14          public String creditCard() {...}
F.15  }
```

Figure 5.10: Test Definition and Fixture after “Add Column” Refactoring of Expected-Value Column

The “Add Column” refactoring is applied to add an expected-value column to the test definition. The Figure 5.10 shows the changed test definition and fixture (the changes are highlighted in bold). The test definition has a new column *credit card()* in A.2 and the fixture has a new public parameter less method *creditCard()* with return type *String* in F.14.

5.5.2.2 Remove Column

Motivation

In contrast to the “Remove Column” refactoring, requirement changes can lead to parameters that are not needed anymore and can be removed. For example, in the credit limit example it might be possible that the membership duration is not considered anymore and thus can be deleted.

Summary

Whenever requirements change and a column based acceptance test has to be adjusted to remove needless given- or expected-value columns, the “Remove Column” refactoring can be used to remove the unnecessary columns.

Mechanics

The “Remove Column” refactoring is carried out in the following way:

1. Remove the chosen column in the test definition.
2. Remove all following cells in this column.
3. If the removed column is a given-value column and
    a. the field to be removed is still in use in at least one method, remove the corresponding public field in the fixture. Additionally, find all methods that have at least one reference of that field. For every method, change the return type to String, comment out the body and add a return statement that returns a string as a reminder.
    b. the field to be removed is not used in any method, remove the corresponding field in the fixture.
4. If the removed column is an expected-value column and
    a. the method to be removed is called in at least one method, remove the corresponding method in the fixture. Additionally, find all methods that have at least one call of that method. For every method, change the return type to String, comment out the body and add a return statement that returns a string as a reminder.
    b. the method to be removed is not used in any method, remove the corresponding method in the fixture.

Example: Remove Expected-Value Column without References

The following example shows an acceptance test definition and the linked fixtures (ColumnFixture) before and after the “Remove Column” refactoring. The test is saved in the storage system as *CreditCondition* and is associated with the fixture *CreditCondition* (see Figure 5.11).

A.1	tests.CreditCondition				
A.2	staff	balance	Months	allow credit()	credit limit()
A.3	false	6000	12	true	3000
A.4	true	3000	12	true	5000

```
F.1  public class CreditCondition extends ColumnFixture {
F.2          public double balance;
F.3          public int months;
F.4          public boolean staff;
```

```
F.5
F.6             public boolean allowCredit() {...}
F.7
F.8             public double creditLimit() {...}
F.9
F.10            private double creditLimitCustomer() {...}
F.11
F.12            private double creditLimitStaff() {...}
F.13    }
```

Figure 5.11: Test Definition and Fixture before "Remove Column" Refactoring of an Expected-Value Column

The "Remove Column" refactoring is applied with *allow credit()* as column to be removed.

A.1	tests.CreditCondition				
A.2	staff	balance	Months	~~allow credit()~~	credit limit()
A.3	false	6000	12	~~true~~	3000
A.4	true	3000	12	~~true~~	5000

```
F.1     public class CreditCondition extends ColumnFixture {
F.2             public double balance;
F.3             public int months;
F.4             public boolean staff;
F.5
F.6             public boolean allowCredit() {...}
F.7
F.8             public double creditLimit() {...}
F.9
F.10            private double creditLimitCustomer() {...}
F.11
F.12            private double creditLimitStaff() {...}
F.13    }
```

Figure 5.12: Test Definition and Fixture after "Remove Column" Refactoring of an Expected-Value Column

The "Remove Column" refactoring is applied to remove the expected-value column from the test definition. The Figure 5.12 shows the changed test definition and fixture (the changes are highlighted in bold). The method *allowCredit()* is not called in any other method (withheld due to space constraints). Therefore, the public parameterless method *allowCredit()* in F.6 has been removed in the fixture as well as the column *allow credit()* in A.2, it has been removed from the test definition.

Example: Remove Given Value Column with References

Besides the expected-value column, also given-value columns could be removed with the "Remove Column" refactoring. The following example (see Figure 5.13) uses the same test definition and fixture shown above to present the effect of removing a given-value column with the "Remove Column" refactoring.

A.1	tests.CreditCondition				
A.2	staff	balance	months	allow credit()	credit limit()
A.3	false	6000	12	true	3000
A.4	true	3000	12	true	5000

```
F.1   public class CreditCondition extends ColumnFixture {
F.2           public double balance;
F.3           public int months;
F.4           public boolean staff;
F.5
F.6           public boolean allowCredit() {
F.7                   if (creditLimit() == 0)
F.8                           return false;
F.9                   return true;
F.10          }
F.11
F.12          public double creditLimit() {
F.13                  if (staff)
F.14                          return creditLimitStaff();
F.15                  else
F.16                          return creditLimitCustomer();
F.17          }
F.18
F.19          private double creditLimitCustomer() {...}
F.20
F.21          private double creditLimitStaff() {...}
F.22  }
```

Figure 5.13: Test Definition and Fixture before "Remove Column" Refactoring with Given Value Column

The "Remove Column" refactoring is applied with *staff* as the column to be removed. The resulting fixture code and test definition can be seen in Figure 5.14.

A.1	tests.CreditCondition				
A.2	~~staff~~	balance	months	allow credit()	credit limit()
A.3	~~false~~	6000	12	true	3000
A.4	~~true~~	3000	12	true	5000

```
F.1   public class CreditCondition extends ColumnFixture {
F.2           public double balance;
F.3           public int months;
F.4           public boolean staff;
F.5
F.6           public String allowCredit() {
F.7                   /* TODO: Needs to be changed */
F.8                   /*
F.9                   if (creditLimit() == 0)
F.10                          return false;
F.11                  return true;
F.12                  */
F.13                  return "This test needs to be changed";
F.14          }
F.15
F.16          public String creditLimit() {
F.17                  /* TODO: Needs to be changed */
F.18                  /*
F.19                  if (staff)
F.20                          return creditLimitStaff();
F.21                  else
F.22                          return creditLimitCustomer();
F.23                  */
F.24                  return "This test needs to be changed";
F.25          }
F.26
F.27          private double creditLimitCustomer() {...}
```

```
F.28
F.29            private double creditLimitStaff() {...}
F.30    }
```

Figure 5.14: Test Definition and Fixture after "Remove Column" Refactoring with Given Value Column

As briefly mentioned in the refactoring catalogue (see Table 5.1), the column *staff* in the test definition as well as the field *staff* (see F.4) in the fixture is removed. As the field is still referenced in the method *creditLimit()* (see F.19), the method body has been commented out, the return type has changed to *String*, a TODO comment has been added and the method returns a string as a reminder (see F.16 to F.25) for the developers and the customer to see that this fixture has to be adjusted. Furthermore, the method *allowCredit()* has also changed (see F.6 to F.14) as it calls the method *creditLimit()* and due to the changed return type of *creditLimit()* it must be edited, too.

tests.CreditCondition			
balance	months	allow credit()	credit limit()
6000	12	true expected This test needs to be changed actual	3000 expected This test needs to be changed actual
3000	12	true expected This test needs to be changed actual	5000 expected This test needs to be changed actual

Figure 5.15: Execution result after "Remove Column" Refactoring with References

Since this refactoring changes the behaviour (see Table 5.1), the test must fail. Figure 5.15 shows the result after executing the refactored test. The test is consistent with the fixture and it fails returning a message explaining the problem. The developers and the customer can immediately see that the acceptance test has been modified and the fixture has to be adjusted manually to restore the expected behaviour.

5.5.3 DoFixture

In the refactoring descriptions for DoFixture the following nomenclature is used:

⇒ Actions are associated to a specific method in the fixture that name matches the camel-cased keywords (first and every odd cell), the number of parameters and the return value Boolean.

5.5.3.1 Rename Action

Motivation

Acceptance test processable by DoFixture are typically used to test workflows or processes. The structure is easily readable and business facing. In other words, workflows can easily be translated into acceptance tests without the need to change the structure. The workflow is built upon single actions that are executed. For example, the workflow could start with gathering the personal information of the applicant, continue to create the bank account and check whether the account has been created properly at last. It also might happen that the nomenclature changes after the acceptance tests have been created or typing errors exist. In this case, the tests have to be updated to be understandable by the customer and all stakeholders.

Summary

Whenever the business nomenclature changes or typing errors are present, the appropriate acceptance tests can be updated with the “Rename Action” refactoring.

Mechanics

The “Rename Action” refactoring is carried out in the following way:

1. Rename all occurrences of the changed action in the test definition
2. Rename the corresponding method of the original action in the fixture to the camel-cased name of the new action.
3. Find all method calls to the renamed method and update the reference with the new name.

Example: Rename Action

The following example shows an acceptance test definition and the linked fixtures (DoFixture) before and after the "Rename Action" refactoring. The test is saved in the storage system as *CreateBankAccountBranch* and is associated with the fixture *CreateBankAccountBranch* (see Figure 5.16).

A.1	fitlibrary.DoFixture			
A.2	start	tests.CreateBankAccountBranch		
A.3	enter first name of applicant	Heiko	and second name	Ordelt
A.4	choose type of account	chequing		
A.5	check that bank account does not exist			
A.6	create bank account			
A.7	check that bank account does exist			
A.8	check that account type is	chequing		

```
F.1   public class CreateBankAccountBranch extends DoFixture {
F.2
F.3           public boolean enterFirstNameOfApplicantAndSecondName(String
      applicant, String name) {...}
F.4
F.5           public boolean chooseTypeOfAccount(String account) {...}
F.6
F.7           public boolean createBankAccount() {...}
F.8
F.9           public boolean checkThatBankAccountDoesExist() {...}
F.10
F.11          public boolean checkThatBankAccountDoesNotExist() {...}
F.12
F.13          public boolean checkThatAccountTypeIs(String is) {...}
F.14  }
```

Figure 5.16: Test Definition and Fixture before "Rename Action" Refactoring

The "Rename Action" refactoring is applied with *choose account type* as new name of the *choose type of account* action.

A.1	fitlibrary.DoFixture			
A.2	start	tests.CreateBankAccountBranch		
A.3	enter first name of applicant	Heiko	and second name	Ordelt
A.4	**choose account type**	chequing		
A.5	check that bank account does not exist			
A.6	create bank account			
A.7	check that bank account does exist			
A.8	check that account type is	chequing		

```
F.1   public class CreateBankAccountBranch extends DoFixture {
F.2
F.3           public boolean enterFirstNameOfApplicantAndSecondName(String
      applicant, String name) {...}
F.4
F.5           public boolean chooseAccountType(String account) {...}
F.6
F.7           public boolean createBankAccount() {...}
F.8
F.9           public boolean checkThatBankAccountDoesExist() {...}
F.10
F.11          public boolean checkThatBankAccountDoesNotExist() {...}
```

```
F.12
F.13            public boolean checkThatAccountTypeIs(String is) {...}
F.14    }
```

Figure 5.17: Test Definition and Fixture after "Rename Action" Refactoring

The Figure 5.17 shows the changed test definition and fixture (the changes are highlighted in bold). All occurrences of the action *choose type of account* (see A.4) have been replaced by the new action *choose account type*. Additionally, the method *chooseTypeOfAccount(String account)* (see F.5) has been renamed to *chooseAccountType(String account)*. If any other method had called the renamed method directly, the appropriate call would have been renamed as well.

5.5.3.2 Add Action

Motivation

As mentioned before, DoFixture is used to translate business processes into executable acceptance tests. It is very likely that process will be extended at some point. For example, to include ordering a credit card after a bank account has been created which might not be apparent from the beginning of the system development.

Summary

Whenever a workflow acceptance test based on a DoFixture has to be extended to be up-to-date with process changes, the appropriate test can be updated with the "Add Action" refactoring.

Mechanics

The "Add Action" refactoring is carried out in the following way:

1. Define an action to insert with the needed keywords and parameters and specify the position.
2. Add the defined action at the specified position to the test definition.
3. Add the appropriate method of the new action to the fixture.

Example: Add Action

The following example shows an acceptance test definition and the linked fixtures (DoFixture) before and after the "Add Action" refactoring. The test is saved in the storage system as

CreateBankAccountBranch and is associated with the fixture *CreateBankAccountBranch* (see Figure 5.18).

A.1	fitlibrary.DoFixture			
A.2	start	tests.CreateBankAccountBranch		
A.3	enter first name of applicant	Heiko	and second name	Ordelt
A.4	choose type of account	chequing		
A.5	check that bank account does not exist			
A.6	create bank account			
A.7	check that bank account does exist			
A.8	check that account type is	chequing		

```
F.1  public class CreateBankAccountBranch extends DoFixture {
F.2
F.3          public boolean enterFirstNameOfApplicantAndSecondName(String
     applicant, String name) {...}
F.4
F.5          public boolean chooseTypeOfAccount(String account) {...}
F.6
F.7          public boolean createBankAccount() {...}
F.8
F.9          public boolean checkThatBankAccountDoesExist() {...}
F.10
F.11         public boolean checkThatBankAccountDoesNotExist() {...}
F.12
F.13         public boolean checkThatAccountTypeIs(String is) {...}
F.14 }
```

Figure 5.18: Test Definition and Fixture before "Add Action" Refactoring

The "Add Action" refactoring is applied with the new action *issue credit card* with one parameter after the action *create bank account*.

A.1	fitlibrary.DoFixture			
A.2	start	tests.CreateBankAccountBranch		
A.3	enter first name of applicant	Heiko	and second name	Ordelt
A.4	choose type of account	chequing		
A.5	check that bank account does not exist			
A.6	create bank account			
A.6.1	**issue credit card**	**TODO**		
A.7	check that bank account does exist			
A.8	check that account type is	chequing		

```
F.1  public class CreateBankAccountBranch extends DoFixture {
F.2
F.3          public boolean enterFirstNameOfApplicantAndSecondName(String
     applicant, String name) {...}
F.4
F.5          public boolean chooseTypeOfAccount(String account) {...}
F.6
F.7          public boolean createBankAccount() {...}
F.8
F.9          public boolean checkThatBankAccountDoesExist() {...}
F.10
F.11         public boolean checkThatBankAccountDoesNotExist() {...}
F.12
F.13         public boolean checkThatAccountTypeIs(String is) {...}
```

```
F.14
F.15            public boolean issueCreditCard(String card) {
F.16                    return false;
F.17            }
F.18    }
```

Figure 5.19: Test Definition and Fixture after "Add Action" Refactoring

The Figure 5.19 shows the changed test definition and fixture (the changes are highlighted in bold). The new action *issue credit card* with one parameter has been added to the test definition after the *create bank account* action (see A.6.1). Furthermore, a corresponding method *issueCreditCard(String card)* has been added to the fixture (see F.15 to F.17). It returns the value false as it is has not been implemented yet.

5.5.3.3 Remove Action

Motivation

In contrast to the "Add Action" refactoring, processes can be changed in a way that the workflow is shrunk and thus some actions of a workflow acceptance test can be removed.

Summary

Whenever a workflow acceptance test based on a DoFixture has to be adjusted to be up-to-date with process changes, the appropriate test can be updated with the "Remove Action" refactoring.

Mechanics

The "Remove Action" refactoring is carried out in the following way:

1. Remove the chosen action at the specified position from the test definition.
2. If no occurrence of the removed action is left in the test definition
    a. and the corresponding method of the action is not called within the fixture remove the method.
    b. and the corresponding method of the action is called within the fixture find all methods that reference the method. For every method, change the return type to *String*, comment out the body and add a return statement that returns a string as a reminder.

Example: Remove Action without Method References

The following example shows an acceptance test definition and the linked fixtures (DoFixture) before and after the "Remove Action" refactoring without method references. The test is saved in the storage system as *CreateBankAccountBranch* and is associated with the fixture *CreateBankAccountBranch* (see Figure 5.20).

A.1	fitlibrary.DoFixture			
A.2	start	tests.CreateBankAccountBranch		
A.3	enter first name of applicant	Heiko	and second name	Ordelt
A.4	choose type of account	chequing		
A.5	check that bank account does not exist			
A.6	create bank account			
A.7	check that bank account does exist			
A.8	check that account type is	chequing		

```
F.1   public class CreateBankAccountBranch extends DoFixture {
F.2
F.3           public boolean enterFirstNameOfApplicantAndSecondName(String
      applicant, String name) {...}
F.4
F.5           public boolean chooseTypeOfAccount(String account) {...}
F.6
F.7           public boolean createBankAccount() {...}
F.8
F.9           public boolean checkThatBankAccountDoesExist() {...}
F.10
F.11          public boolean checkThatBankAccountDoesNotExist() {...}
F.12
F.13          public boolean checkThatAccountTypeIs(String is) {...}
F.14  }
```

Figure 5.20: Test Definition and Fixture before "Remove Action" Refactoring without Method References

The "Remove Action" refactoring is applied with action *check that bank account does not exist* in line A.5 to be removed.

A.1	fitlibrary.DoFixture			
A.2	start	tests.CreateBankAccountBranch		
A.3	enter first name of applicant	Heiko	and second name	Ordelt
A.4	choose type of account	chequing		
A.5	~~check that bank account does not exist~~			
A.6	create bank account			
A.7	check that bank account does exist			
A.8	check that account type is	chequing		

```
F.1   public class CreateBankAccountBranch extends DoFixture {
F.2
F.3           public boolean enterFirstNameOfApplicantAndSecondName(String
      applicant, String name) {...}
F.4
F.5           public boolean chooseTypeOfAccount(String account) {...}
F.6
F.7           public boolean createBankAccount() {...}
F.8
F.9           public boolean checkThatBankAccountDoesExist() {...}
F.10
F.11          ~~public boolean checkThatBankAccountDoesNotExist() {...}~~
```

```
F.12
F.13            public boolean checkThatAccountTypeIs(String is) {...}
F.14    }
```

Figure 5.21: Test Definition and Fixture after "Remove Action" Refactoring without Method References

The Figure 5.21 shows the changed test definition and fixture (the changes are highlighted in bold). The action *check that bank account does not exist* without a parameter has been removed from the test definition after the *choose type of account* action (see A.5). The method *checkThatBankAccountDoesNotExist()* (see F.11) is not called in any other method (withheld due to space constraints) and thus has been removed from the fixture.

Example: Remove Action with Method References

The following example shows an acceptance test definition and the linked fixtures (DoFixture) before and after the "Remove Action" refactoring with method references. The test is saved in the storage system as *CreateBankAccountBranch* and is associated with the fixture *CreateBankAccountBranch* (see Figure 5.22).

A.1	fitlibrary.DoFixture			
A.2	start	tests.CreateBankAccountBranch		
A.3	enter first name of applicant	Heiko	and second name	Ordelt
A.4	choose type of account	chequing		
A.5	check that bank account does not exist			
A.6	create bank account			
A.7	check that bank account does exist			
A.8	check that account type is	chequing		

```
F.1     public class CreateBankAccountBranch extends DoFixture {
F.2
F.3             public boolean enterFirstNameOfApplicantAndSecondName(String
        applicant, String name) {...}
F.4
F.5             public boolean chooseTypeOfAccount(String type) {...}
F.6
F.7             public boolean createBankAccount() {...}
F.8
F.9             public boolean checkThatBankAccountDoesExist() {
F.10                    return !checkThatBankAccountDoesNotExist();
F.11            }
F.12
F.13            public boolean checkThatBankAccountDoesNotExist() {
F.14                    /* ... checking existence ... */
F.15            }
F.16
F.17            public boolean checkThatAccountTypeIs(String is) {...}
F.18    }
```

Figure 5.22: Test Definition and Fixture before "Remove Action" Refactoring with Method References

The "Remove Action" refactoring is applied with the action *check that bank account does not exist* to be removed.

A.1	fitlibrary.DoFixture			
A.2	start	tests.CreateBankAccountBranch		
A.3	enter first name of applicant	Heiko	and second name	Ordelt
A.4	choose type of account	chequing		
A.5	~~**check that bank account does not exist**~~			
A.6	create bank account			
A.7	check that bank account does exist			
A.8	check that account type is	chequing		

```
F.1   public class CreateBankAccountBranch extends DoFixture {
F.2
F.3           public boolean enterFirstNameOfApplicantAndSecondName(String
      applicant, String name) {...}
F.4
F.5           public boolean chooseTypeOfAccount(String type) {...}
F.6
F.7           public boolean createBankAccount() {...}
F.8
F.9           public String checkThatBankAccountDoesExist() {
F.10                  /* TODO: Needs to be changed */
F.11                  /*
F.12                  return !checkThatBankAccountDoesNotExist();
F.13                  */
F.14                  return "This test needs to be changed";
F.15          }
F.16
F.17          public boolean checkThatBankAccountDoesNotExist() {
F.18                  /* ... checking existence ... */
F.19          }
F.20
F.21          public boolean checkThatAccountTypeIs(String is) {...}
F.22  }
```

Figure 5.23: Test Definition and Fixture after "Remove Action" Refactoring with Method References

The Figure 5.23 shows the changed test definition and fixture (the changes are highlighted in bold). The action *check that bank account does not exist* has been removed (see A.5). Since the method *checkThatBankAccountDoesNotExist()* (see F.17 to F.19) is referenced in the method *checkThatBankAccountDoesExist()* (see F.9 to F.15) it is commented out and according to the refactoring catalogue (see Table 5.2) the return value is changed to *String* and a reminder message is returned.

6 Implementation of Automated Refactoring Tool Support

6.1 Environment of Implementation

6.1.1 Eclipse Platform

Eclipse is an open-source software framework written primarily in Java. In its default form, it is an Integrated Development Environment (IDE) for Java developers, consisting of the Java Development Tools (JDT). Figure 6.1 shows the three main layers of Eclipse:

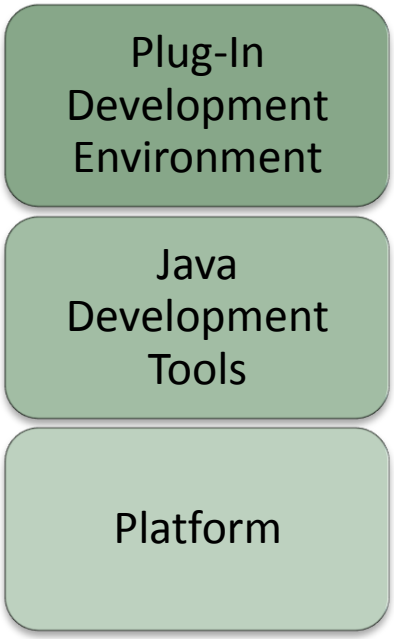

Figure 6.1: The three Layers of Eclipse (Gamma, et al., 2003 p. 5)

⇒ **Platform**

The Plug-In Development Environment (PDE) extends the JDT with support for developing plug-ins.

⇒ **Java Development Tools (JDT)**

The Java development tools add a full featured Java IDE to Eclipse.

⇒ **Platform**

The Eclipse platform defines a common programming language-neutral infrastructure.

The platform consists of several key components that are layered into a user interface (UI)-independent core and a UI layer, as shown in Figure 6.2.

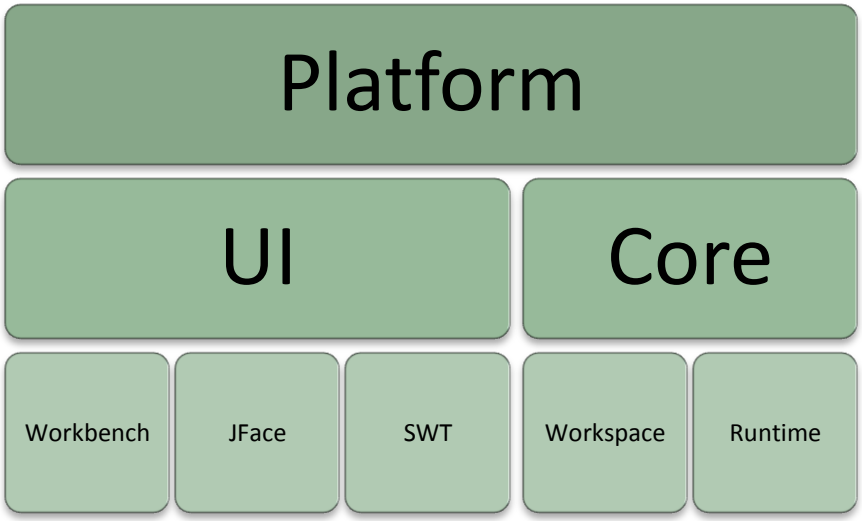

Figure 6.2: Eclipse Platform Overview (Gamma, et al., 2003 p. 6)

⇒ **Runtime**

The run-time component defines the plug-in infrastructure. It discovers the available plug-ins on start-up and manages the plug-in loading.

⇒ **Workspace**

A workspace manages one or more top-level projects. A project consists of file and folders that map onto the underlying file system.

⇒ **Standard Widget Toolkit (SWT)**

The SWT provides graphics and defines a standard set of widgets.

⇒ **JFace**

A set of smaller UI frameworks built on top of SWT supporting common UI tasks.

⇒ **Workbench**

The workbench defines the Eclipse UI paradigm. It centers between editors, views and perspectives.

Users can extend the capabilities by installing plug-ins written for the Eclipse software framework, such as development toolkits for other programming languages and can write and contribute their own plug-in modules.

6.1.2 FitClipse

FitClipse (ASE, 2008) is an Eclipse plug-in developed by the ASE Group of the University of Calgary supporting the creation, modification and execution of acceptance tests using the FIT framework and FitLibrary (see 4.2.2.1). Figure 6.3 shows FitClipse running in an Eclipse environment.

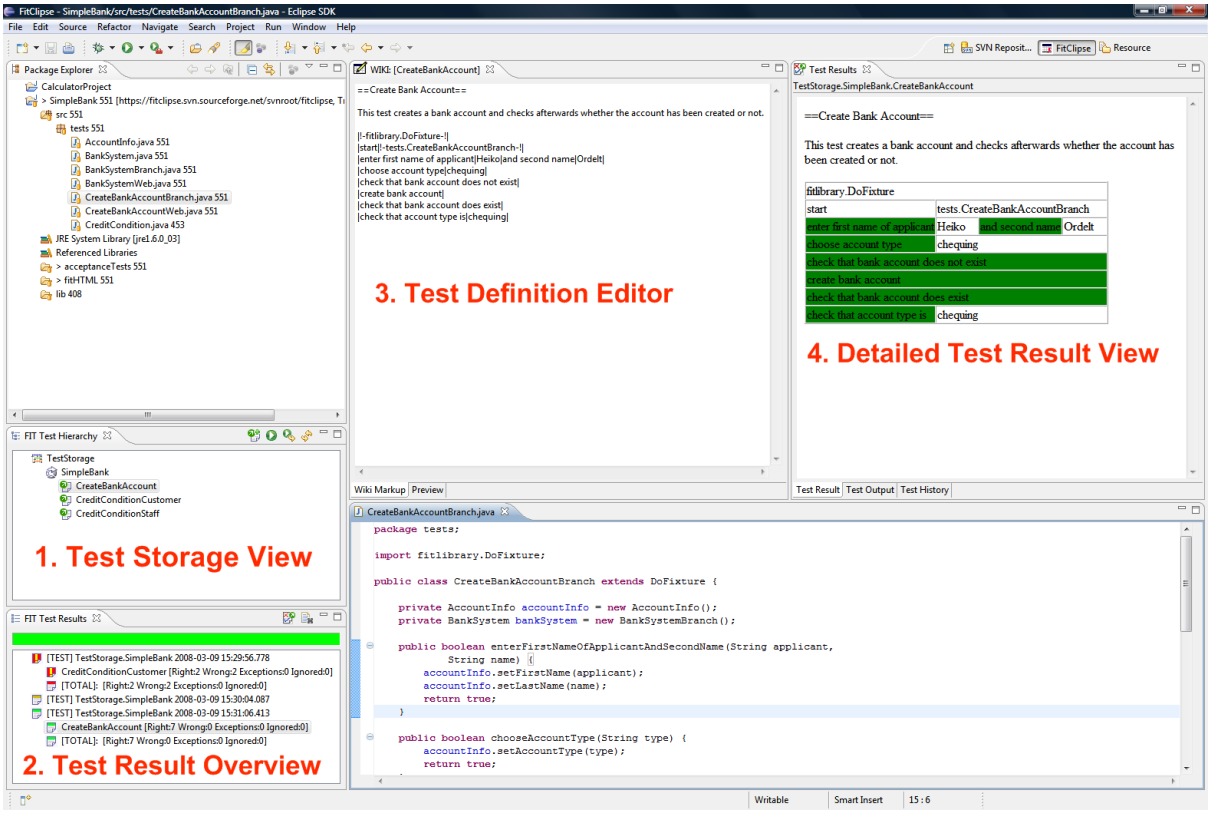

Figure 6.3: FitClipse Overview

The FitClipse user interface is built on different Eclipse views as follows:

⇒ **Test storage view**

The test storage view shows a hierarchy of the test structure of the current project. Tests are shown with a unique name and can be organized in suites.

⇒ **Test result overview**

When tests are executed, the result of the test run is shown in this view. It can contain multiple test results and distinguishes between unimplemented failures (never been green before) and regression failures (green at least once).

⇒ **Test definition editor**

To create and edit the test definition of acceptance tests the included editor can be used. It is a simple text editor showing the test definition and allowing the user to change and save the definition of a specific test.

⇒ **Detailed test result view**

In addition to the general test result overview, FitClipse can also show details of a specific test run in the test result view. The view shows the feedback from the output of the FIT framework.

While FitClipse was originally developed to use FitNesse to store acceptance tests and test results, the dependency of FitNesse has recently been removed and replaced with a XML based persistence layer. Besides the main functionality of managing and running acceptance tests, it also supports Executable Acceptance Test Driven Development by providing an automated fixture generation feature.

6.2 Used Eclipse Plug-ins

6.2.1 Java Development Tools

JDT stands for Java Development Tools, the sub-project of the Eclipse project that develops tools for programming in Java. The JDT subproject is broken down into components. Each component operates like a project on its own, with its dedicated set of committers, bug categories and mailing lists (Eclipse, 2008).

The following components belong to the Java Development Tools:

⇒ **APT**

Java 5.0 annotation processing infrastructure.

⇒ **Core**

Java IDE headless infrastructure.

⇒ **Debug**

Debug support for Java.

⇒ **Text**

Java editing support.

⇒ **UI**

Java IDE User Interface.

The JDT project provides the tool plug-ins that implement a Java IDE supporting the development of any Java application, including Eclipse plug-ins. It adds a Java project nature and Java perspective to the Eclipse Workbench as well as a number of views, editors, wizards, builders and code merging and refactoring tools. The JDT project allows Eclipse to be a development environment for itself (Eclipse, 2008).

6.2.2 Language Toolkit

At the EclipseCon conference in 2004, a great deal of interest sparked the idea of adding more generic language IDE infrastructure to Eclipse. Many people have been impressed by the powerful functionality in the Eclipse Java tooling and would like to be able to leverage that support in other languages (Eclipse, 2007).

Therefore, the Eclipse Foundation started to extract some of the JDT functionality into a generic layer. In Eclipse 3.0, the generic parts of the JDT refactoring infrastructure were put into a general IDE layer. This common programming-language tooling layer is called the Eclipse Language Toolkit (LTK). It provides an infrastructure for language-independent refactoring in two packages:

⇒ **org.eclipse.ltk.core.refactoring**

The core package provides classes and interfaces for refactoring operations and a mechanism to allow third parties to participate in a refactoring.

⇒ **org.eclipse.ltk.ui.refactoring**

The UI package allows users to utilize some basic UI components for refactoring wizards.

This infrastructure is a logical starting point for writing refactoring support for languages other than Java. The two important classes of the refactoring framework that define the base functionality are the Refactoring class and the Change class.

The Refactoring class represents the entire refactoring lifecycle, including precondition checks, generating the set of changes and post-condition checks. The Change class itself performs more expensive validation on the input to determine whether the refactoring is appropriate and performs the workspace modifications induced by it. A Change instance can also encapsulate an undo for another change, allowing the user to back out of a refactoring after it has completed (Eclipse, 2007).

6.3 Integration into the Eclipse Refactoring Framework

The FitClipse refactoring extension has been integrated in the Eclipse framework by using the Java Development Tools (JDT) and the Language Toolkit (LTK). The test storage view of FitClipse has been extended with a context menu that allows the user to first select an acceptance test and then select the refactoring task he wants to apply (see Figure 6.4).

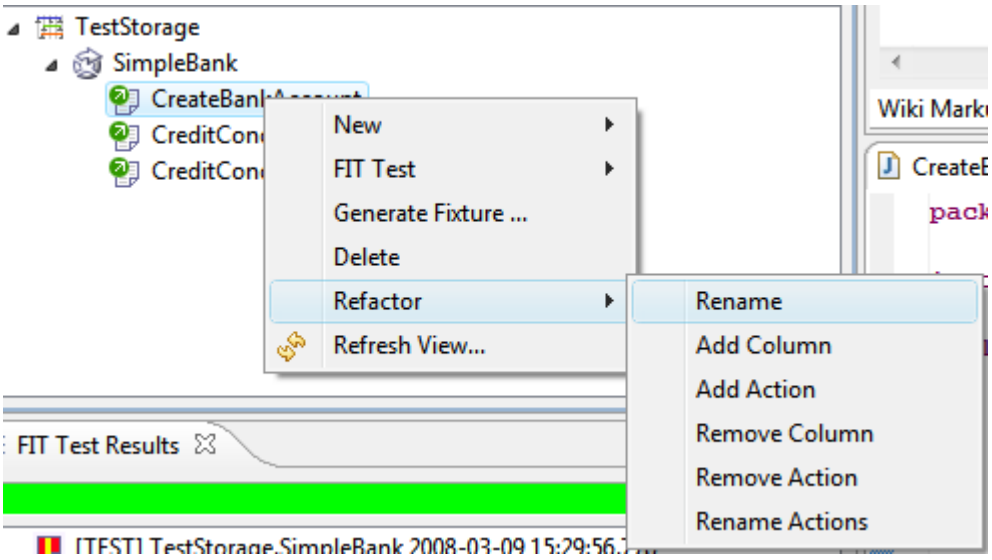

Figure 6.4: Refactoring Menu of FitClipse

When the desired refactoring has been chosen, an input mask gives the user the ability to insert all input data needed for the appropriate refactoring. For example, Figure 6.5 shows the input mask of the “Rename Acceptance Test” refactoring which consists of a simple textbox.

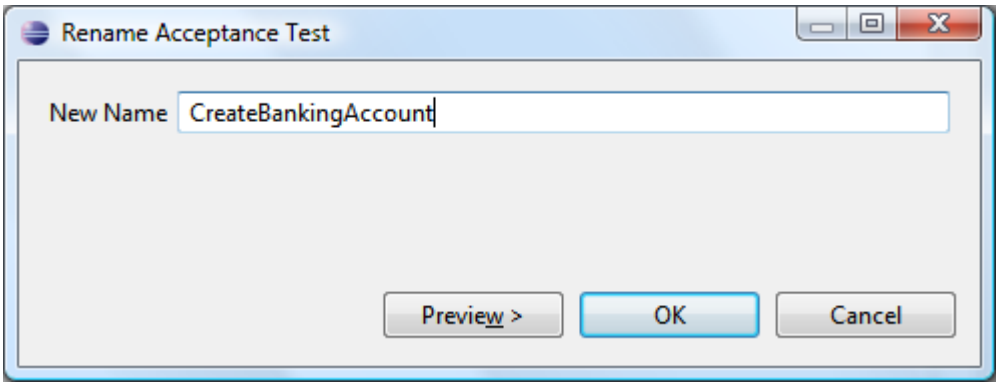

Figure 6.5: Implemented User Interface of Rename Refactoring (Input)

The user has to enter the new name and after clicking on “Preview” the preview shows the resulting changes to test definition and fixture code (see Figure 6.6).

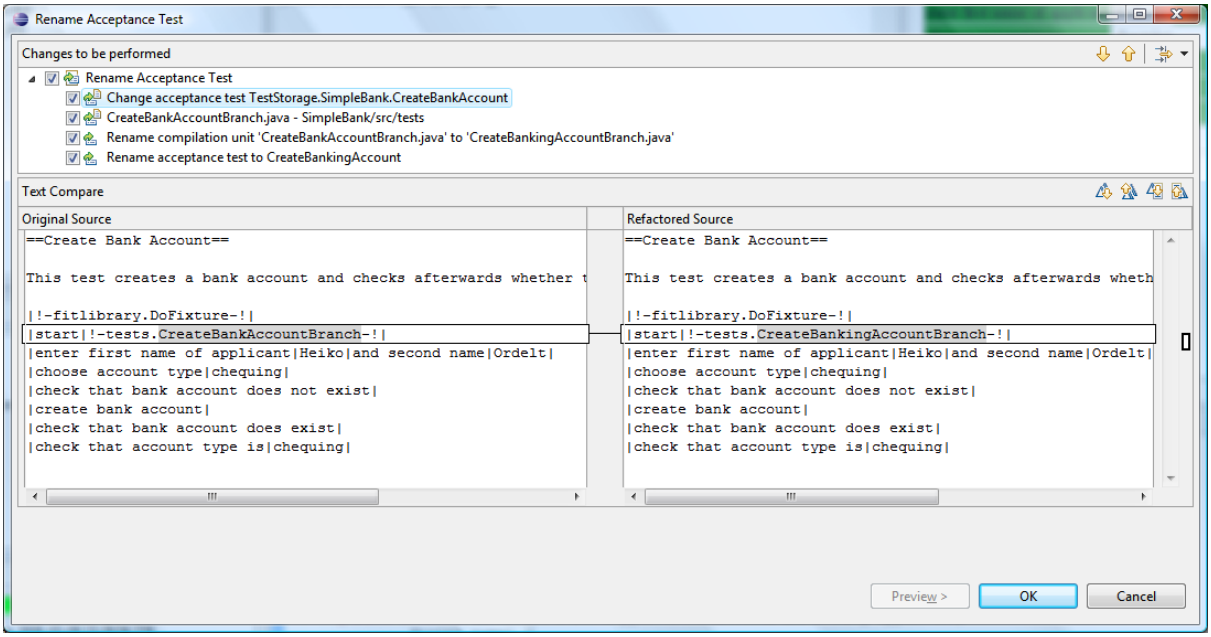

Figure 6.6: Implemented User Interface of Rename Refactoring (Preview)

The preview is provided by the Eclipse framework and is exactly the same preview that is used for the common source code refactoring. In the upper part, all single changes (e.g. changes to test definition, changes to fixture etc.) are displayed. Every change can be selected which updates the bottom part showing a comparison of the affected element before and after the refactoring. However, some changes like renaming a Java file have no comparison. Additionally, the preview has the capability to disable changes which will be ignored at the end. After clicking on the “OK” button, the modifications will be applied immediately.

6.4 Overall Structure

6.4.1 Package Structure

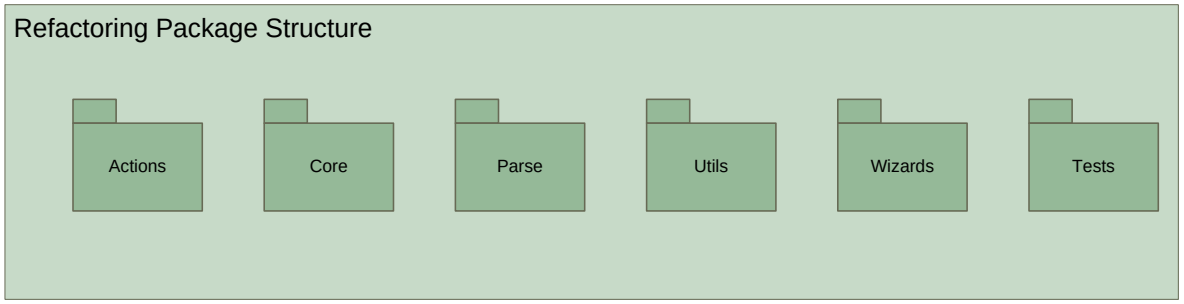

Figure 6.7: FitClipse Refactoring Package

The refactoring extension is separated into several packages:

The action package contains classes that build the connection between the FitClipse user interface and the refactoring interface. When a user selects a refactoring task to be applied, the correct class in the actions package is called triggering the refactoring process. All non-user interface classes that are working with the refactoring framework are placed in the core package. It contains the refactoring and processor classes. The parse package provides all needed parser like fixture parser and test definition parser as well as the classes that include the fixture specific refactoring implementations. Some helper classes used in the refactoring process are in the utils package. The wizards package contains the refactoring wizard classes. Among others, this includes the main wizard class that controls the workflow and the corresponding input page to allow the user to specify the refactoring input. The tests package contains the unit and acceptance tests of the refactoring extension. The unit tests are also used by the ASE group's continuous integration server that builds and checks the FitClipse several times a day.

6.4.2 Architecture and Design

The refactoring extension was built by using the LTK and JDT plug-ins included in the Eclipse SDK.

Figure 6.8 shows an architectural overview of the whole application including the main components.

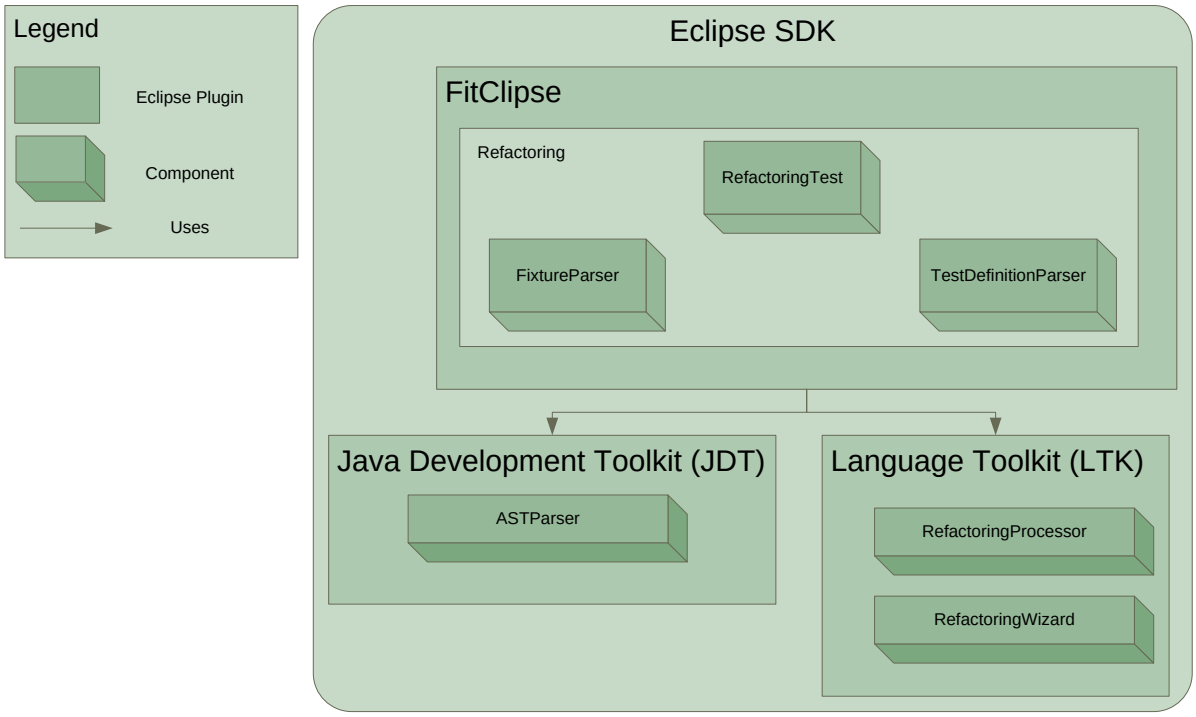

Figure 6.8: Refactoring Extension Architecture Overview

It can be seen that FitClipse is running as a plug-in in an Eclipse environment. Due to the scope of this work, only the refactoring package is shown which contains the three main components RefactoringTest, FixtureParser and TestDefinitionParser. They will be discussed and explained further later on in this chapter. FitClipse uses the Java Development Tools (JDT) and Language Toolkit (LTK) plug-ins provided by Eclipse. The main component used in JDT is the ASTParser that is able to parse Java source code. In the LTK plug-in, the RefactoringProcessor, responsible for the refactoring lifecycle, and the RefactoringWizard, for the user interface support, are used. Next, the main components of FitClipse will be described.

6.4.3 Core Components

6.4.3.1 TestDefinitionParser

The test definition parser component consists of the ITestDefinitionParser interface as well as three classes that represent cells, rows and tables (see Figure 6.9).

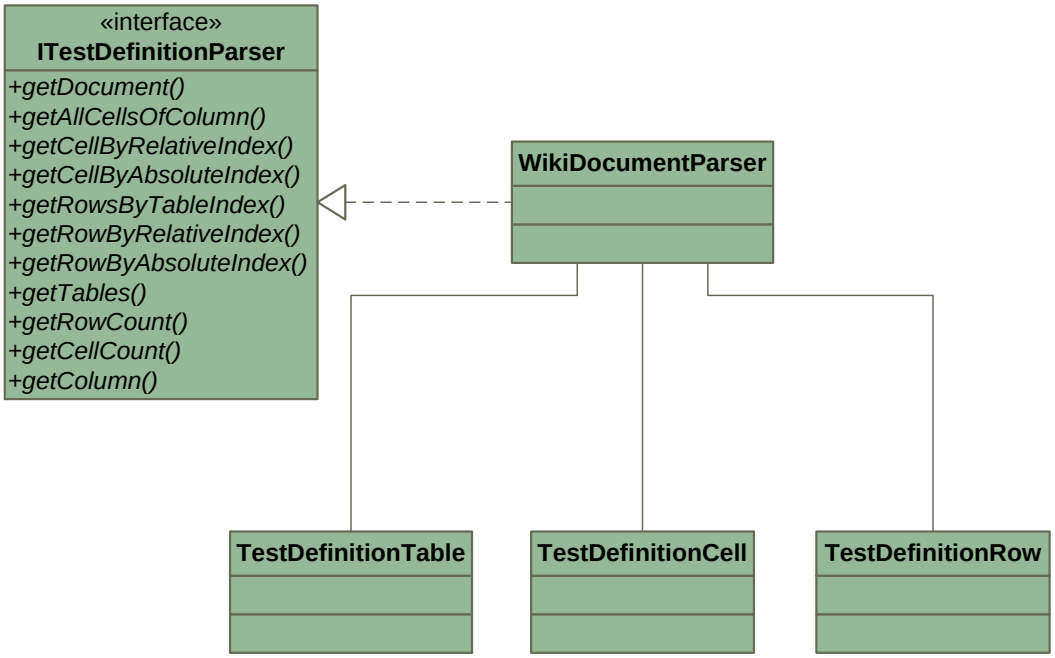

Figure 6.9: Class Diagram of TestDefinitionParser

The purpose of the test definition parser is to process a test definition and to divide the tables into different syntactical elements like cells, rows and tables Currently, the only ITestDefinitionParser interface implementation provided is the WikiDocumentParser that is able to process wiki code introduced by FitNesse. However, due to the defined interface it is easy to add support for more input formats like HTML by simply implementing the interface.

The parser works with absolute and relative indexes. An absolute index of an element like a cell or row is defined by its position compared to the beginning of the first table. The relative index of a cell represents the position in a specific row. Along with the cell index, the relative index of a row is defined by the position in a specific table. Figure 6.10 shows two test definitions that have the same structure. The only difference is that the test definition on the left side is split into two tables while test definition 2 is one whole table. It can be seen that the absolute index refers to the same rows even if the test definition is split into several tables. This allows the parser to handle multiple test definitions that have the same structure but are different in their table layout.

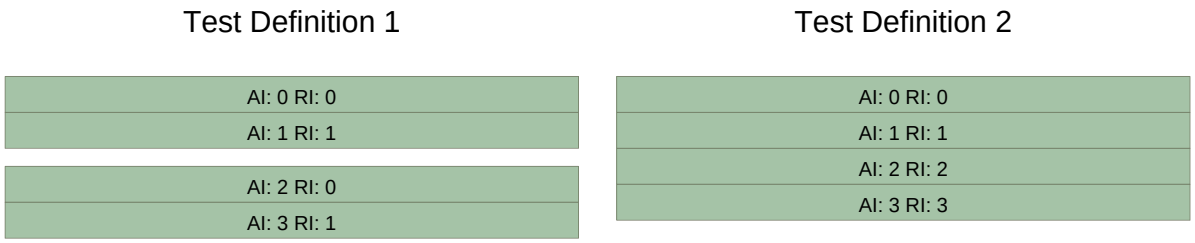

Figure 6.10: TestDefinitionParser Indexing Operation Mode

The extracted information is used to present the structure of the test definition in the refactoring user interface as well as to provide the Eclipse refactoring framework with the needed information to carry out the appropriate refactoring changes to the test definition. The TestDefinitionParser works completely on its own without any usage of the Eclipse framework. Only regular expressions and lists are used to provide the needed functionality.

6.4.3.2 FixtureParser

The fixture parser component consists of the IFixtureParser interface that defines the mandatory functionality for the fixture handling and modification. FitClipse has currently one implementing class FixtureASTParser that relies on the ASTParser provided by the Eclipse framework. Its purpose is to modify the linked fixtures of acceptance tests when they get refactored as well as analyzing the structure. For example, removing a method or checking if a specific method exists. Figure 6.11 shows all relevant classes and interfaces and their relationships in a class diagram.

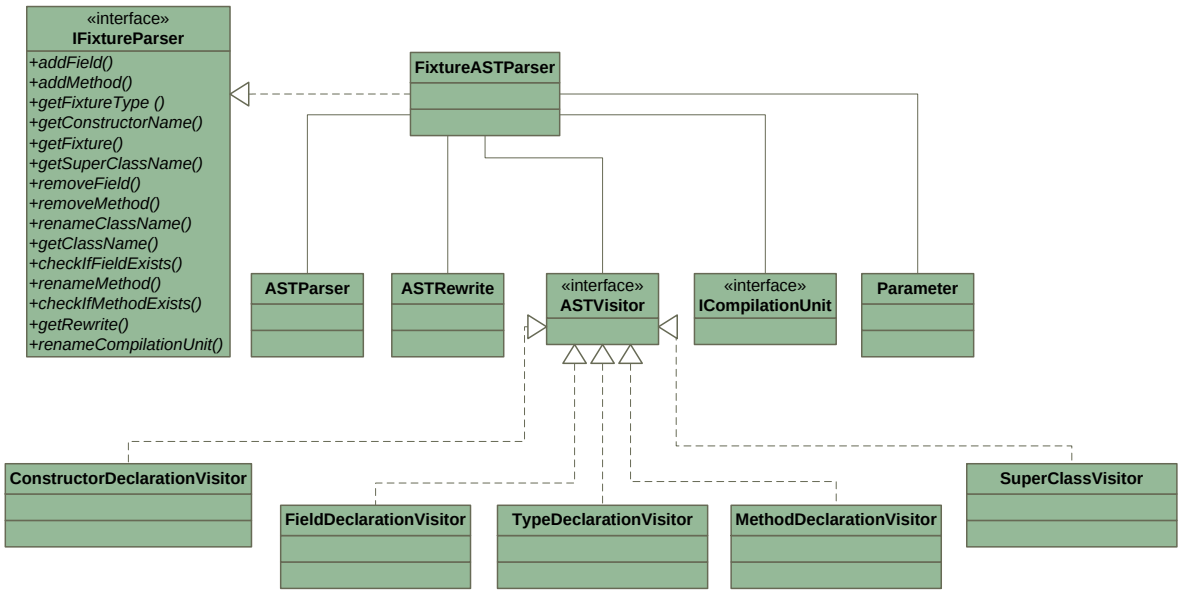

Figure 6.11: Class Diagram of FixtureParser

The ASTParser is provided by the Eclipse framework and is able to build an abstract syntax tree (AST) of Java source code. The resulting AST is comparable to the DOM tree model of an XML file. Just like DOM, the AST allows to modify the tree model and reflects these modifications in the Java source code. The interface ICompilationUnit represents an entire Java compilation unit (source file) which is typically the fixture file in this application.

The AST uses the Visitor design pattern which purpose is to perform computations on traversals through data structures (Kastens, et al., 2007 p. 68). To find various elements like fields or methods in the AST, the ASTVisitor interface has to be implemented. The refactoring extension includes several implementations of the ASTVisitor interface like MethodDeclarationVisitor (to find methods) or SuperClassVisitor (to find a super class). They are called for every node of the AST while traversing through it and return the search element when it is found. See Figure 6.12 for an overview of the typical usage of the AST.

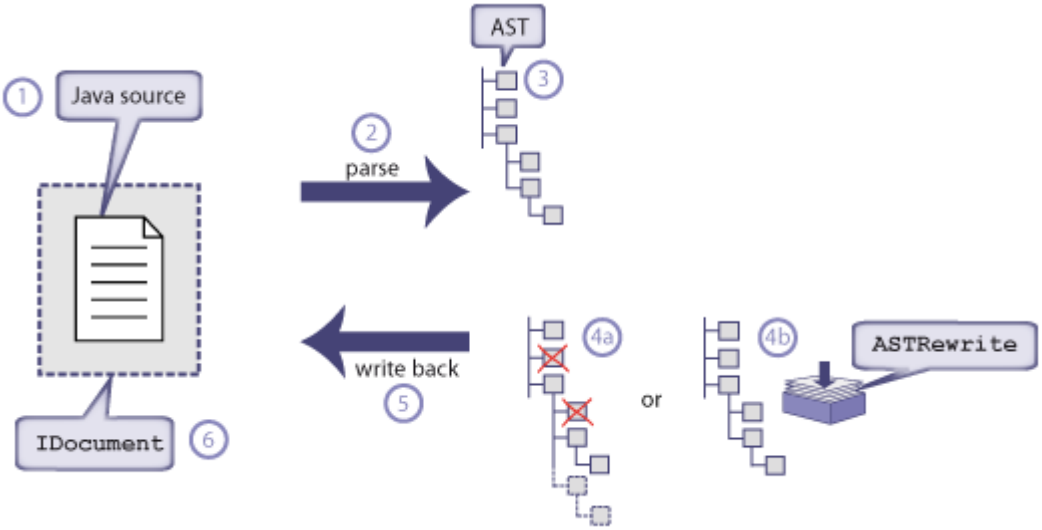

Figure 6.12: Abstract Syntax Tree Workflow (Kuhn, et al., 2006)

First, the source code (given as a Java file or as string) is parsed by the ASTParser and an AST is built. Afterwards, this AST can be modified either directly or with the help of the ASTRewrite class. It collects descriptions of modifications applied to nodes of the AST and translates these descriptions into text changes that can then be applied to the original source.

6.4.3.3 *RefactoringTest*

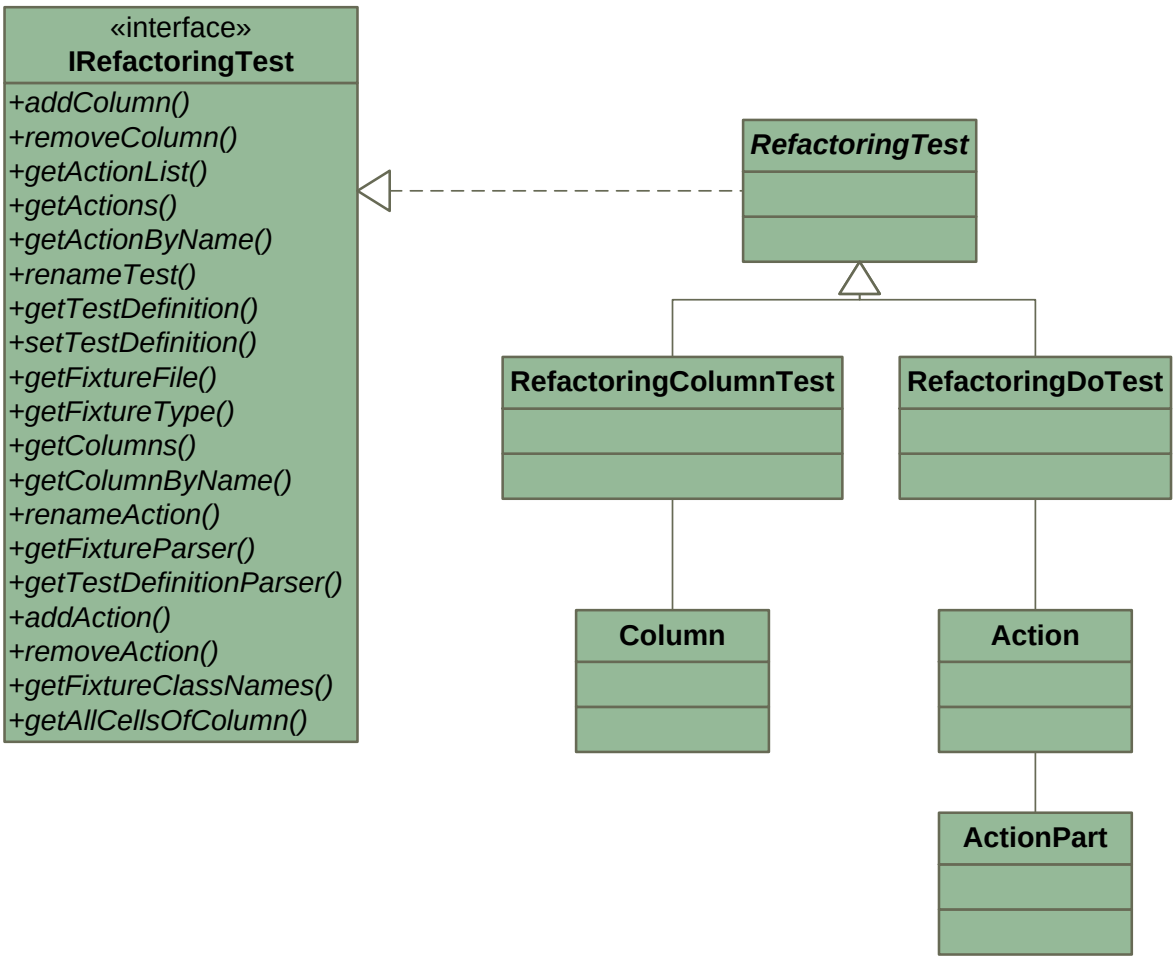

Figure 6.13: Class Diagram of RefactoringTest

The interface IRefactoringTest encapsulates a simple acceptance test and defines the functionality that is needed for the various refactoring types. The abstract class RefactoringTest implements the common functionality that is independent of the actual test type. Since FitClipse currently supports ColumnFixture and DoFixture, the appropriate classes are extending the RefactoringTest class.

The RefactoringColumnTest class uses the Column class that is a simple container to store information about a column. Additionally, the RefactoringDoTest class uses the Action and ActionPart classes to save data about the actions in the test definition.

6.4.3.3.1 RefactoringTestFactory

The RefactoringTestFactory uses the Factory design pattern (see class diagram Figure 6.14). Since the calling class does not know which type of RefactoringTest it must instantiate, the factory class is used.

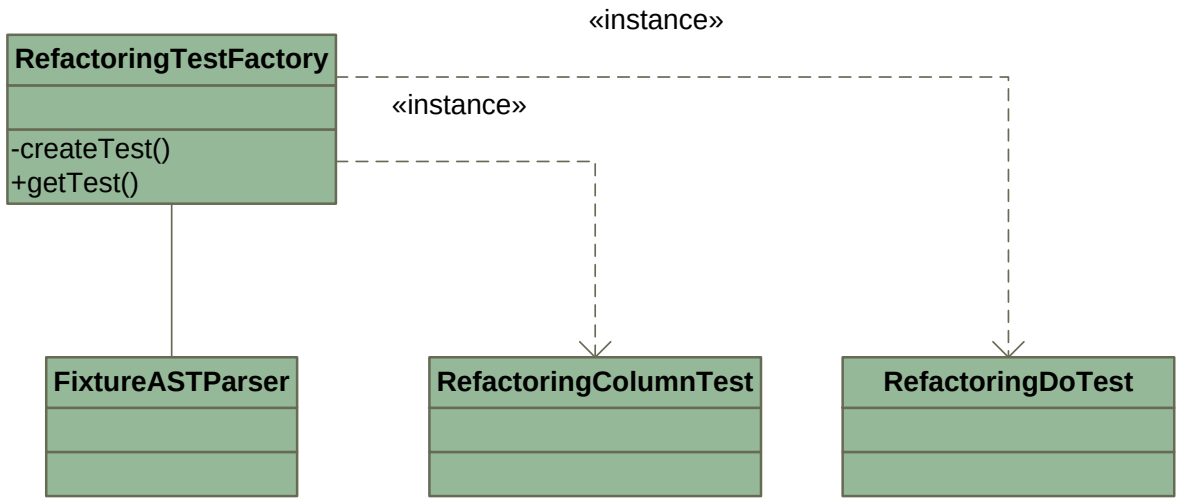

Figure 6.14: Class Diagram of RefactoringTestFactory

Its purpose is to take a generic test object that contains the test definition and a list of all linked fixtures among other information and to return this object as the correct RefactoringTest instance. This is needed because the test definition cannot be used to distinguish between different test types. Figure 6.15 shows an acceptance test processed by a RowFixture. In this case, each cell in the prime column represents a key that identifies one of the records that is expected to be returned (in this case, a prime number).

fitnesse.fixtures.PrimeNumberRowFixture
prime
2
3
5
7
11

```
public class PrimeNumberRowFixture extends RowFixture {
  public Object[] query() throws Exception {
    PrimeData[] array = new PrimeData[5];
    array[0] = new PrimeData(11);
    array[1] = new PrimeData(5);
    array[2] = new PrimeData(3);
    array[3] = new PrimeData(7);
    array[4] = new PrimeData(2);
    return array;
  }
```

Figure 6.15: RowFixture Test and Fixture Code

In addition, Figure 6.16 shows a simple acceptance test processed by a ColumnFixure. A comparison of both tests shows that it is not possible to distinguish between RowFixture and ColumnFixture based on the structure of the test definition.

eg.Division		
numerator	denominator	quotient?
10	2	5
12.6	3	4.2
100	4	33

```
public class Division extends ColumnFixture {
  public double numerator, denominator;
  public double quotient() {
    return numerator/denominator;
  }
}
```

Figure 6.16: ColumnFixture Test and Fixture Code

Therefore, the only way to determine the test type is to analyze the fixture code and especially the super class. The workflow of the test type recognition is shown in Figure 6.17 in detail.

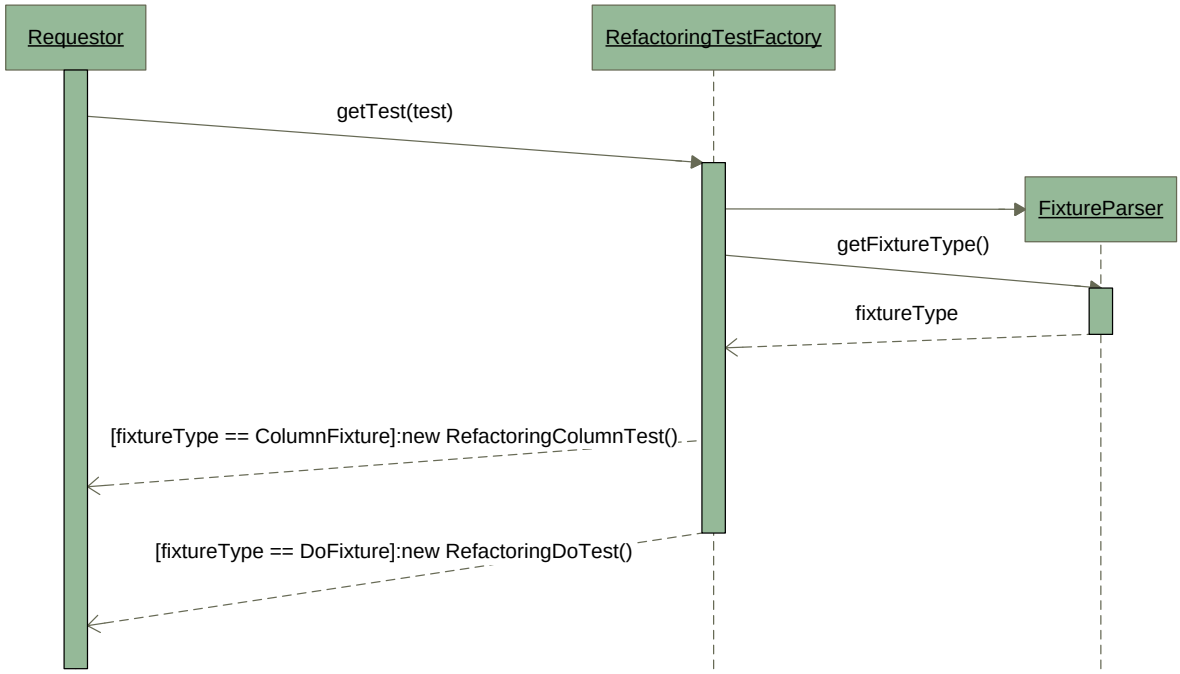

Figure 6.17: RefactoringTestFactory Sequence Diagram

Since the RefactoringTestFactory is called from various objects, the Requestor object is a generic representative for all objects that actually use the Factory. In the first step, the Requestor object calls the RefactoringTestFactory providing a test object of the test to be recognized. The RefactoringTestFactory uses the FixtureASTParser to get the fixture type. This is simply done by getting the super class of the fixture and returning this value as the fixture type. The RefactoringTestFactory returns the appropriate RefactoringTest object based on the test type information.

6.5 Refactoring Execution Workflow

The Eclipse refactoring framework (LTK) follows a specific workflow to perform a refactoring. The general workflow is shown in Figure 6.18.

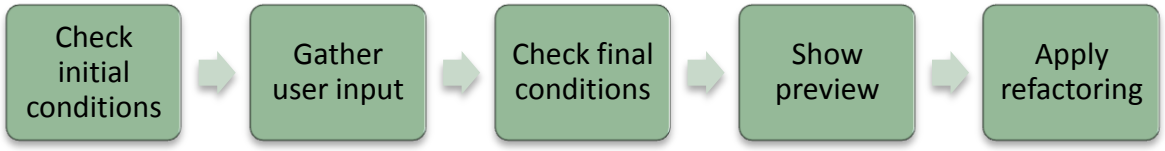

Figure 6.18: Refactoring Workflow

At the beginning of every refactoring task, the initial conditions are checked. This typically includes basic activation inspections like confirming the consistent state of the workspace. The next step is to gather the user input for the respective task. This can vary from being a request for simple to request for complex input data. After all information has been provided, the refactoring framework investigates if the refactoring can be carried out with the given parameter. A preview shows the possible changes which gives the user the ability to see what will change and to gain confidence that the changes match his expectations. As a last step, the changes will be carried out and the refactoring process has finished.

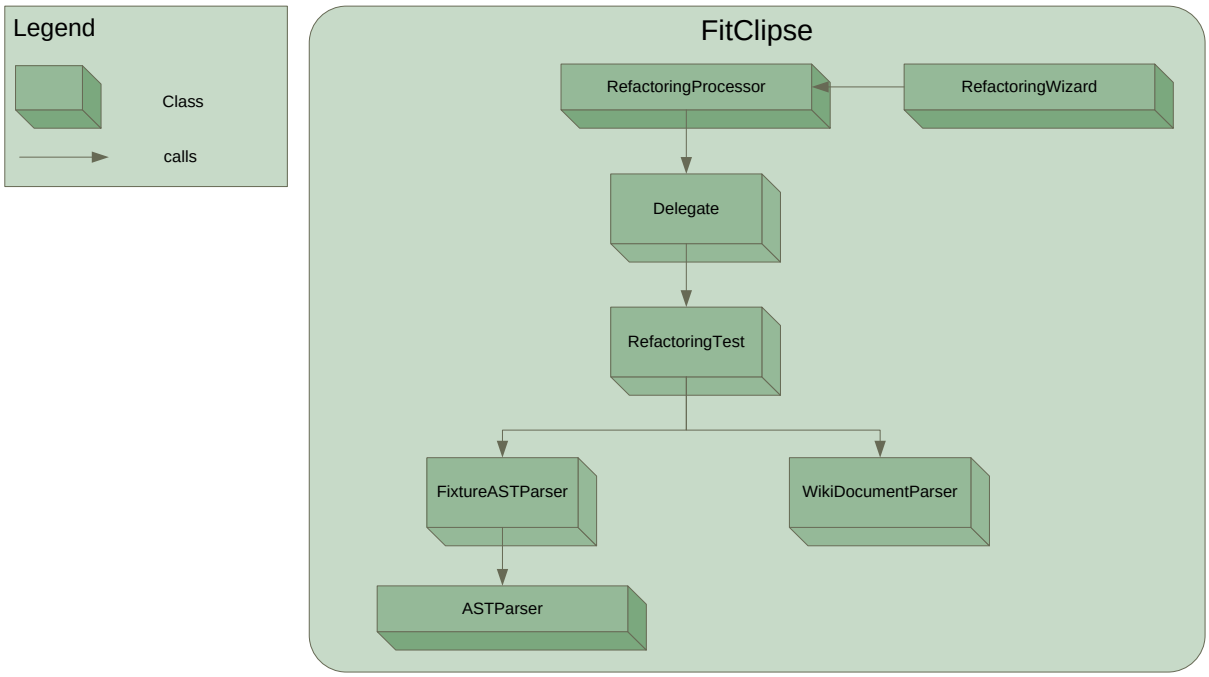

Figure 6.19: Diagram of Refactoring Calling Hierarchy

Figure 6.19 shows all used classes and their relationship during a full refactoring process. The RefactoringProcessor is the link between the RefactoringWizard and the classes that perform the condition checking.

The whole refactoring process is controlled by the RefactoringWizard. Depending on the current step, it calls methods of the RefactoringProcessor class to get the result of the initial condition check or the *Change* objects needed to build the preview. The Delegate class forwards only the method calls from RefactoringProcessor to the RefactoringTest, which depends on the test type RefactoringDoTest or RefactoringColumnTest. These classes use the FixtureASTParser and WikiDocumentParser to create the changes. A detailed view of the workflow is given in Figure 6.20 as a sequence diagram. The objects of the “Add Column” refactoring can be understood as representatives for every other refactoring task.

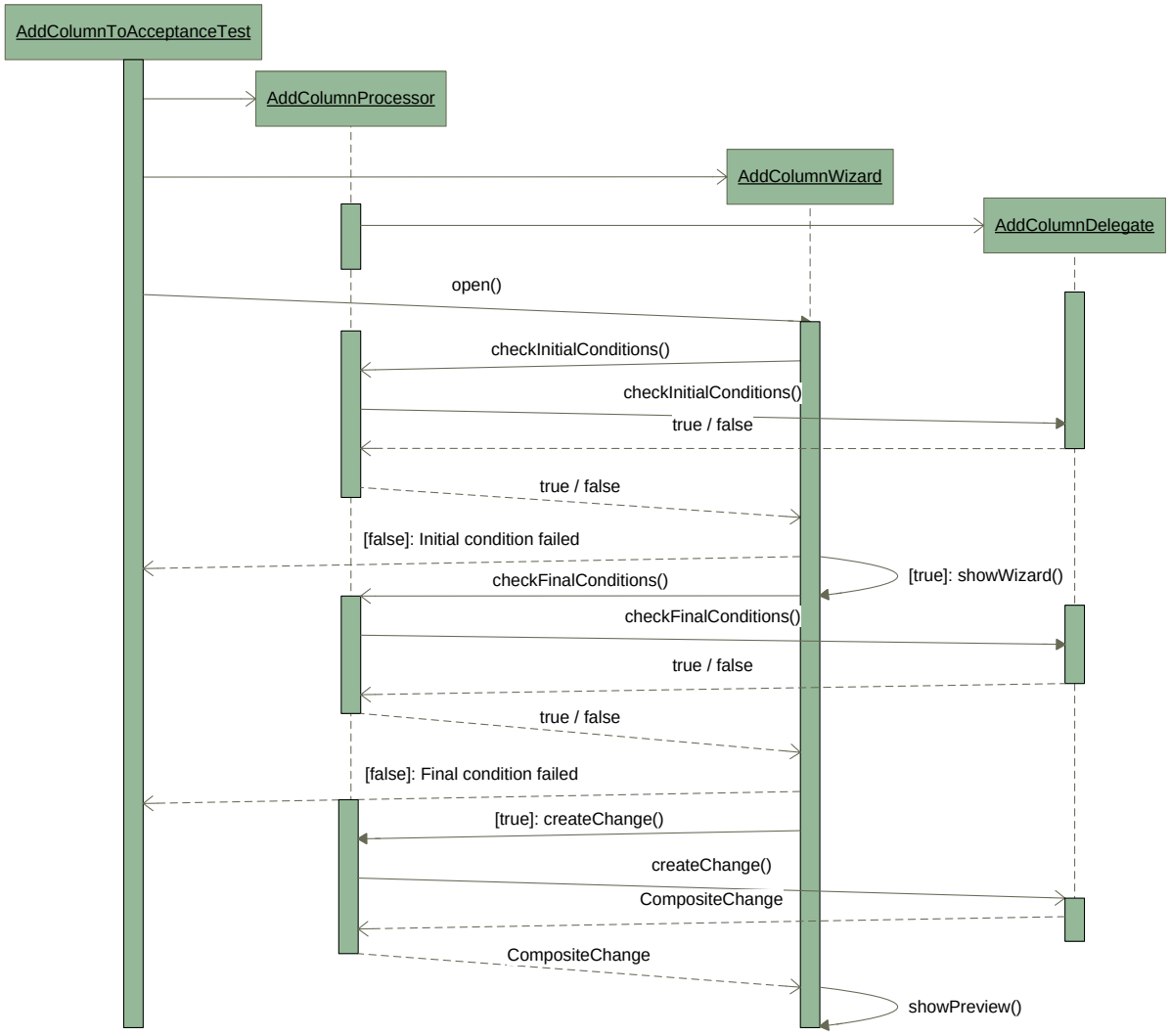

Figure 6.20: Sequence Diagram of the Eclipse Refactoring Workflow

The AddColumnToAcceptanceTest class is triggered by the FitClipse user interface after selecting a refactoring task. First, it creates the needed objects AddColumnProcessor, AddColumnWizard and AddColumnDelegate. Next, it starts the AddColumnWizard that takes the control over the refactoring process and initiates test for the initial conditions. This task is forwarded to the AddColumnDelegate that is returning whether it was successful or not. If it was successful, the AddColumnWizard shows the input mask and gathers the refactoring input from the user. Afterwards, the system makes sure that all changes are consistent. When the test for the initial conditions fails, the responsibility is given back to the AddColumnToAcceptanceTest class, which gives the user a graphical notification with a message explaining why the refactoring failed. If the final test passes, the AddColumnDelegate creates a set of Change objects describing the changes to be made during the refactoring. The AddColumnWizard uses this information to present a preview.

All changes that take place after the preview are executed by the Eclipse refactoring framework and cannot be controlled.

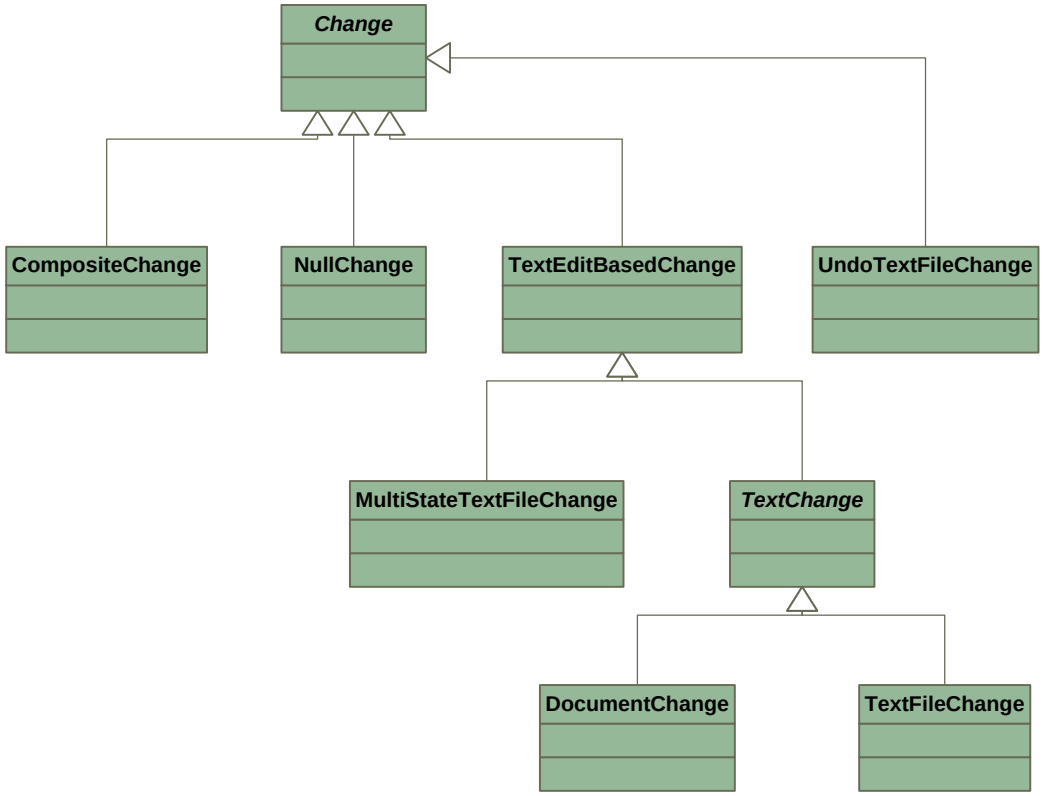

Figure 6.21: Class Diagram of Change Object Hierarchy

The generated Change objects can be of various types. The refactoring extension mostly uses TextFileChange to modify the fixture code and DocumentChange for modifications on the test definition.

6.6 Multiple Test and Fixture Support

As mentioned in 4.2.3, FitClipse supports multi-modal test execution which means that one acceptance test is linked with multiple fixtures. It is also possible that two acceptance tests use the same fixture. Figure 6.22 presents an example test structure. It consists of three acceptance tests T1, T2 and T3 as well as four fixtures F1, F2, F3 and F4. The tests T1 and T2 are both linked with fixture F2.

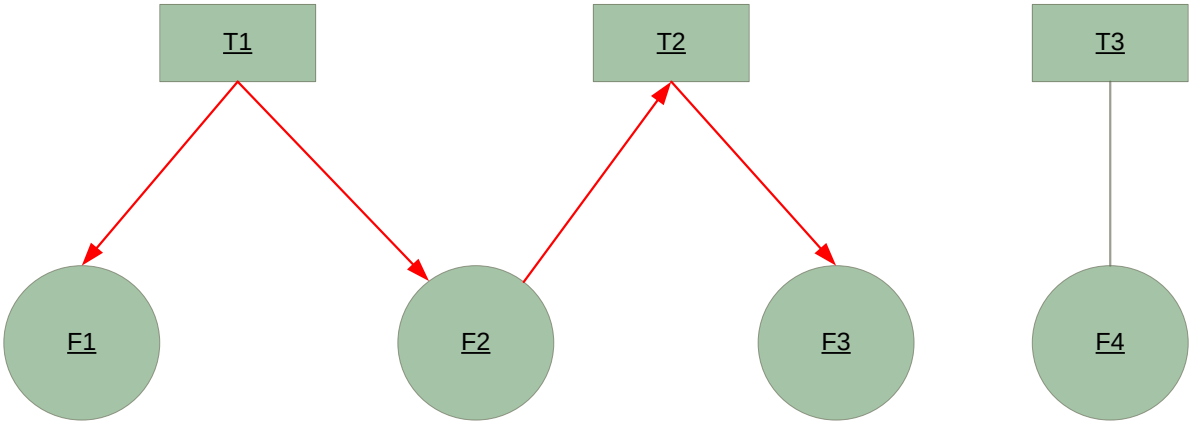

Figure 6.22: Test Database Structure Example

When acceptance test T1 is refactored and fixture F2 is changed, T2 also has to be adjusted. If T2 would not be adjusted, it would be inconsistent with fixture F2. Furthermore, in this case F3 is also affected and needs to be altered to be consistent with the acceptance test T2.

However, this example shows a special case where multiple tests and fixtures can be affected by one single refactoring task. The refactoring extension supports this case by analyzing the test structure and finding all linked fixtures.

It uses the FixtureManager class that was already available. The class is responsible to manage the relationships between the fixtures and acceptance tests of a project. After providing an acceptance test, it returns a list of all related tests and fixtures which are supposed to be part of this modification. Instead of making the modifications of a refactoring to a single acceptance test and fixture, it will be applied to all acceptance tests and fixtures returned by the FixtureManager class.

6.7 Specific Refactoring Implementation

In this chapter, every refactoring implementation is described in detail. The following description structure is used:

- ⇒ Name
- ⇒ Low-fidelity prototype
- ⇒ Look and functioning of user interface mask
- ⇒ Implementation specifics

The name of the refactoring task refers to the refactoring implementation and description given in chapter 5. After the name, the low-fidelity prototype is shown that was used to discuss the design

before starting with the implementation. This leads to the user interface mask that was built in FitClipse that is presented and described in its functioning. Rather than describing implementation details, only specific details like additional features or special procedures are discussed.

6.7.1 Rename Acceptance Test

The “Rename Acceptance Test” refactoring activity implements the appropriate refactoring described in 5.5.1. After selecting the task for an acceptance test in the test storage view, a user interface mask (see low-fidelity prototype in Figure 6.23) is displayed.

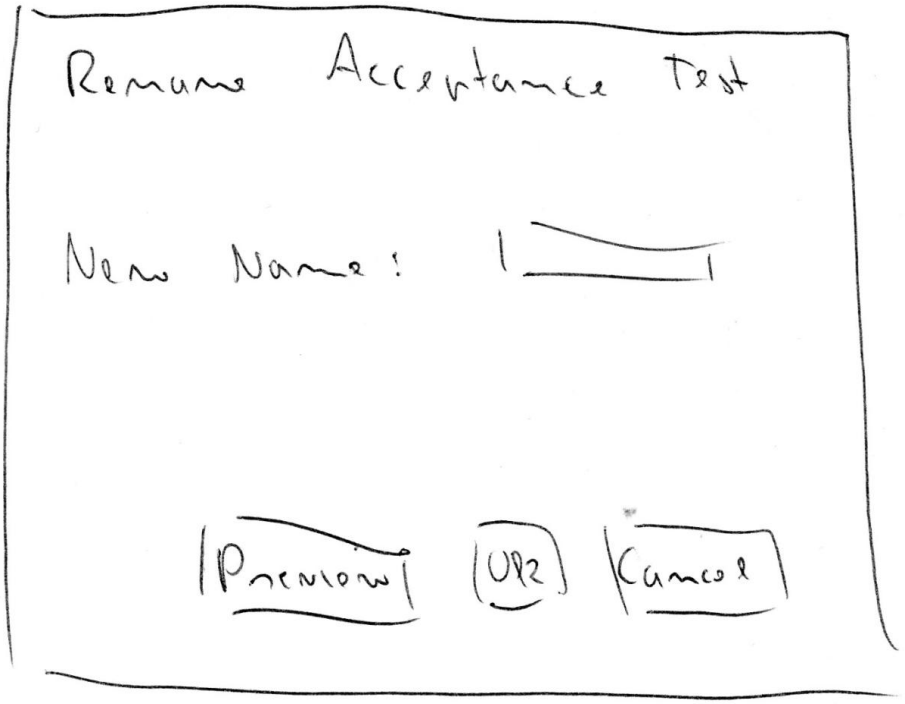

Figure 6.23: Low Fidelity Prototype of "Rename Acceptance Test" Refactoring

The transfer from the low-fidelity prototype to the actual UI wizard can be seen in Figure 6.24. The user interface mask has one text field that allows the user to change the name of the respective acceptance test.

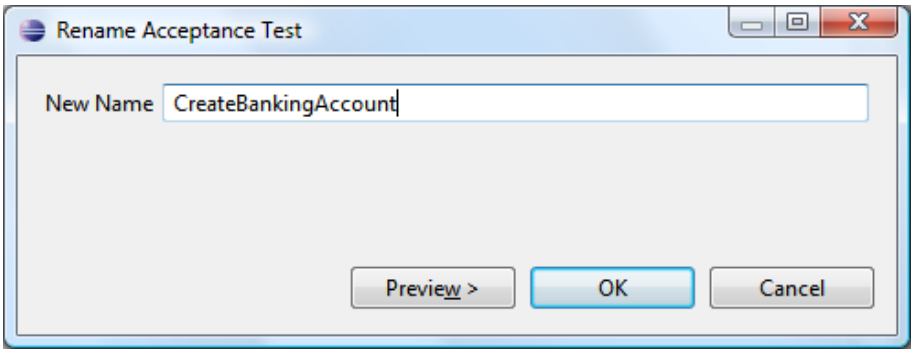

Figure 6.24: Input User Interface Mask of "Rename Acceptance Test" Refactoring

There are two constraints regarding valid input:

⇒ **Different new name**

The new name must be different to the old name to be valid. Additionally, it must start with an uppercase character as the resulting fixture names must follow the valid Java class naming conventions.

⇒ **Resulting fixtures do not exist**

If at least one filename for the new Java classes does already exist in the workspace the input value is not valid.

Implementation Specifics

The “Rename Acceptance Test” refactoring implementation automatically renames multiple fixtures linked with an acceptance test to keep the relationship based on the name. When the name of the acceptance test is changed, all fixtures with the matching file name are renamed as well. The following example shows how the support for multiple fixtures works.

It is assumed that there is an acceptance test *CreateBankAccount* which is linked to the fixtures *CreateBankAccountBranch.java* and *CreateBankAccountWeb.java*. Both fixtures have the same prefix *CreateBankAccount*. When the acceptance test name is changed to *CreateBankingAccount*, the prefix *CreateBankAccount* of both fixtures has to change as well.

Table 6.1: Fixture Rename Procedure Examples of “Rename Acceptance Test” Refactoring

Acceptance Test Name	Fixture Name 1	New Acceptance Test Name	Resulting Fixture Name 1
CreateBankAccount	CreateBankAccountBranch	CreateBankingAccount	CreateBankingAccountBranch
CreateBankAccount	CreateBankAccount	CreateBankingAccount	CreateBankingAccount
CreateBankAccount	BankAccountCreation	CreateBankingAccount	BankAccountCreation

However, when non-prefix changes are made they are not carried out to the fixture names. The Table 6.1 shows some more examples that explain the procedure.

6.7.2 Add Column

The “Add Column” refactoring implements the refactoring description given in 5.5.2.1. After selecting the refactoring task for an acceptance test in the test storage view, the user sees a user interface mask (see low-fidelity prototype in Figure 6.25).

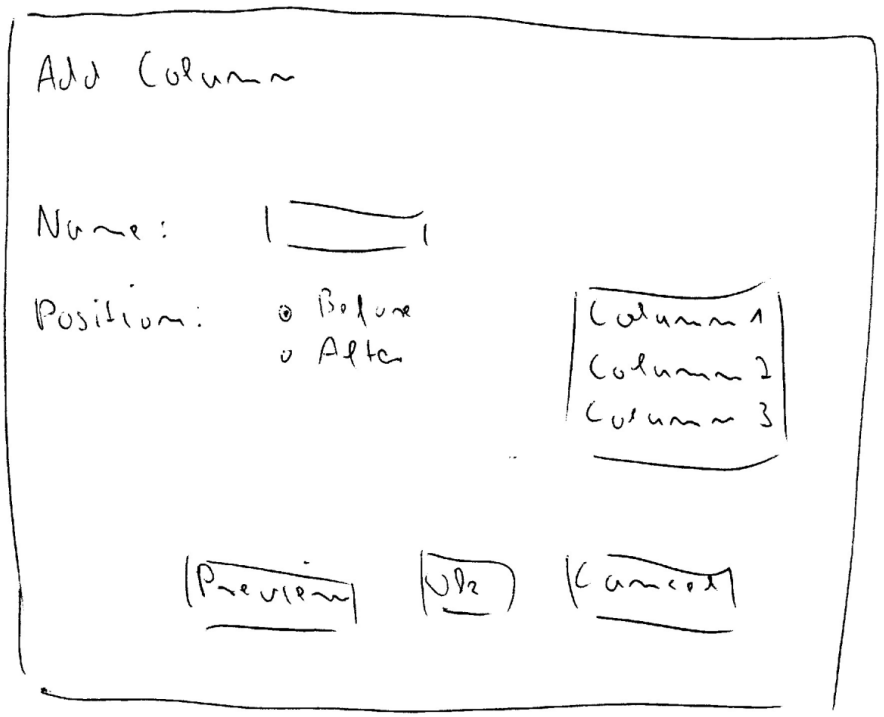

Figure 6.25: Low-fidelity Prototype of "Add Column" Refactoring

The implementation of the low-fidelity prototype can be seen in Figure 6.26. The user interface mask consists of one text field as well as two radio buttons and a select box.

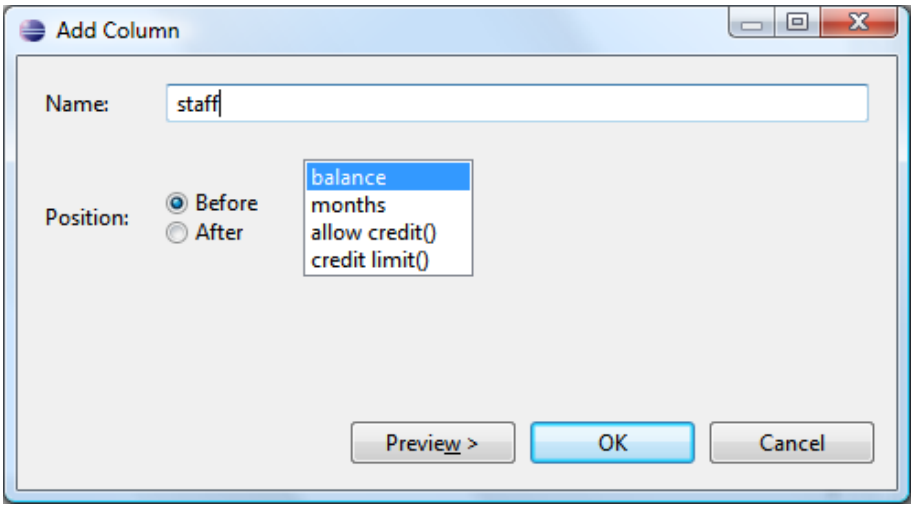

Figure 6.26: Input User Interface Mask of "Add Column" Refactoring

The text field is used to enter the name of the new column. The values “before” or “after” together with a selected existing column the position for the new column.

There are two constraints regarding valid input:

⇒ **New name valid Java identifier**

The new name must be a valid Java field or method identifier since the value is used for the new field/method when the fixture is modified. This is validated with help of the JavaConventions.validateFieldName() and JavaConventions.validateMethodName() methods that are provided by the Eclipse framework.

⇒ **Column does not already exist**

The input gets only accepted if the column to insert does not already exist in the test definition.

Implementation Specifics

The “Add Column” refactoring distinguishes automatically between expected- and given-value columns by analyzing the value. If it ends with an opening and closing bracket, it is considered an expected-value column. In contrast, if it ends with a character it is considered to be a given-value column. This makes it easier for people who are familiar with the FIT framework as it uses the same notation.

If a given-value column is added and the respective field already exists in the fixture, the refactoring will be carried out but without adding a field. This also applies to expected-value column if the connected method already exists.

6.7.3 Remove Column

The “Remove Column” refactoring implements the refactoring description given in 5.5.2.2. After selecting the refactoring for an acceptance test in the test storage view, the user sees a user interface mask (see low-fidelity prototype in Figure 6.27).

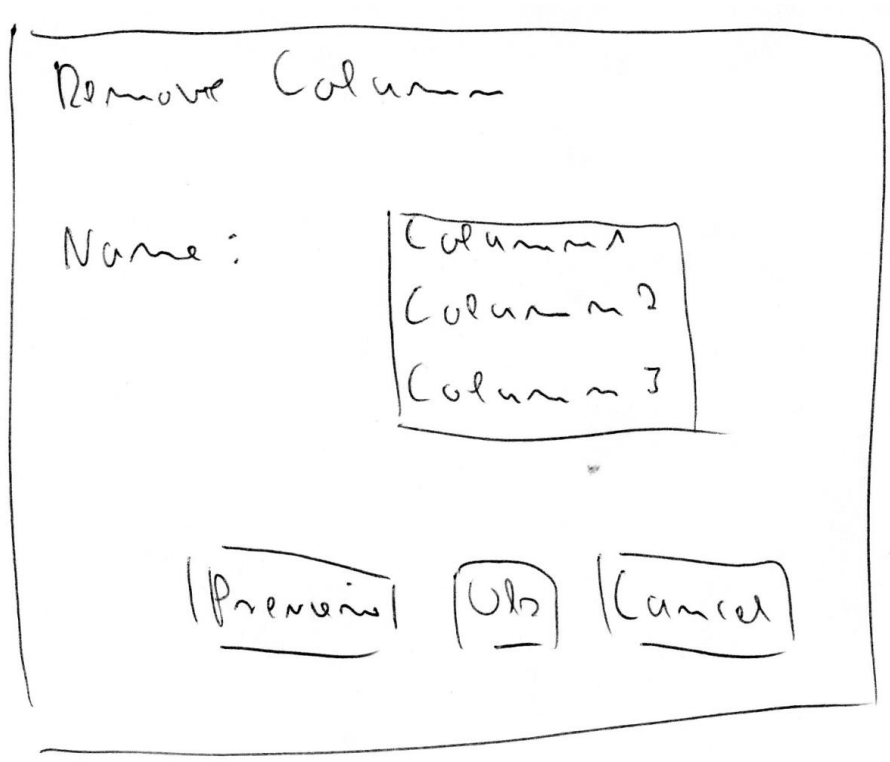

Figure 6.27: Low-Fidelity Prototype of "Remove Column" Refactoring

The implementation of the low-fidelity prototype can be seen in Figure 6.28. The user interface mask consists only of select box including the existing columns of the test definition.

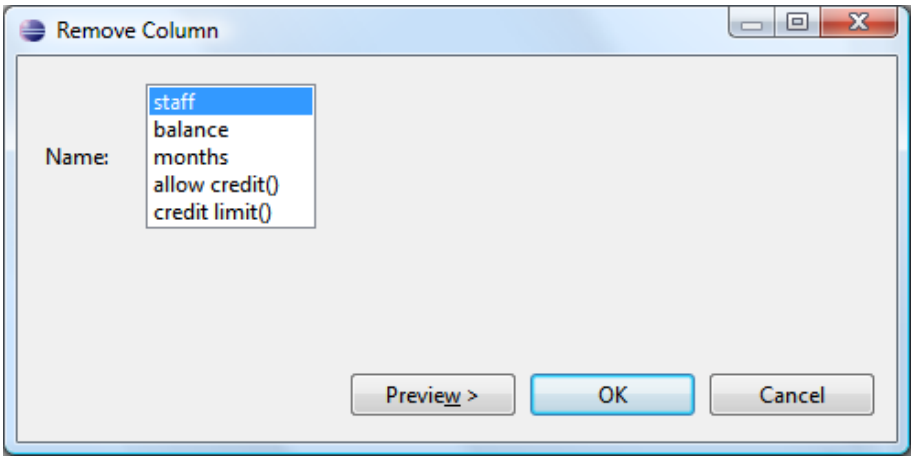

Figure 6.28: Input User Interface Mask of "Remove Column" Refactoring

The user can choose one of the columns to remove it. There is no validation of the input needed.

Implementation Specifics

The “Remove Column” refactoring distinguishes automatically between expected- and given-value columns by analyzing the value. If it ends with an opening and closing bracket, it is considered an expected-value column. In contrast, if it ends with a character it is considered to be a given-value column.

If a given-value column is removed and the connected field does not exist in the fixture anymore, the refactoring task will be completed without removing the field. This also applies to an expected-value column if the representing method does not exist.

6.7.4 Rename Action

The “Rename Action” refactoring implements the refactoring description given in 5.5.3.1. After selecting the refactoring for an acceptance test in the test storage view, the user sees a user interface mask (see low-fidelity prototype in Figure 6.29).

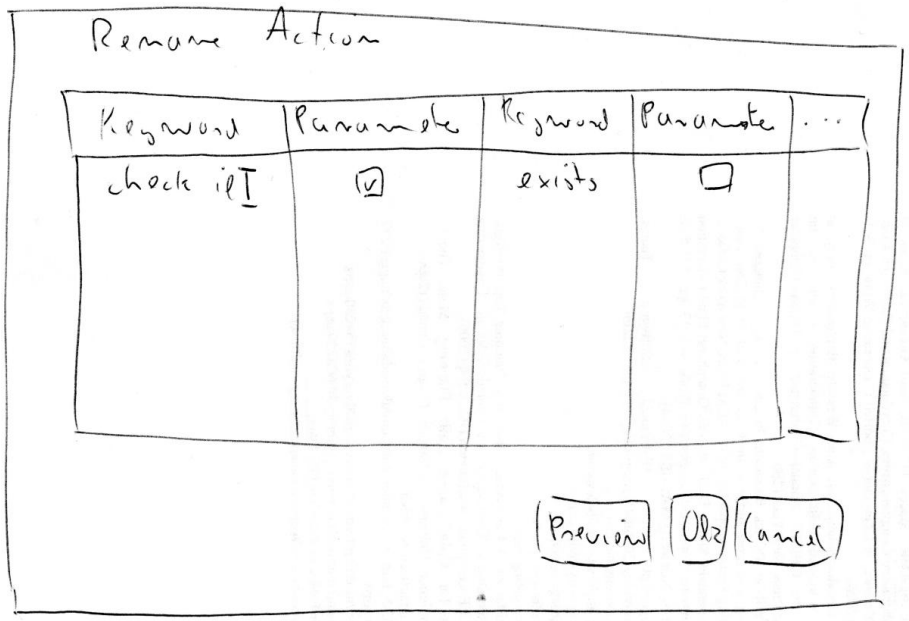

Figure 6.29: Low-Fidelity Prototype of "Rename Action" Refactoring

The implementation of the low-fidelity prototype can be seen in Figure 6.30. The user interface mask consists of a table that shows the existing actions.

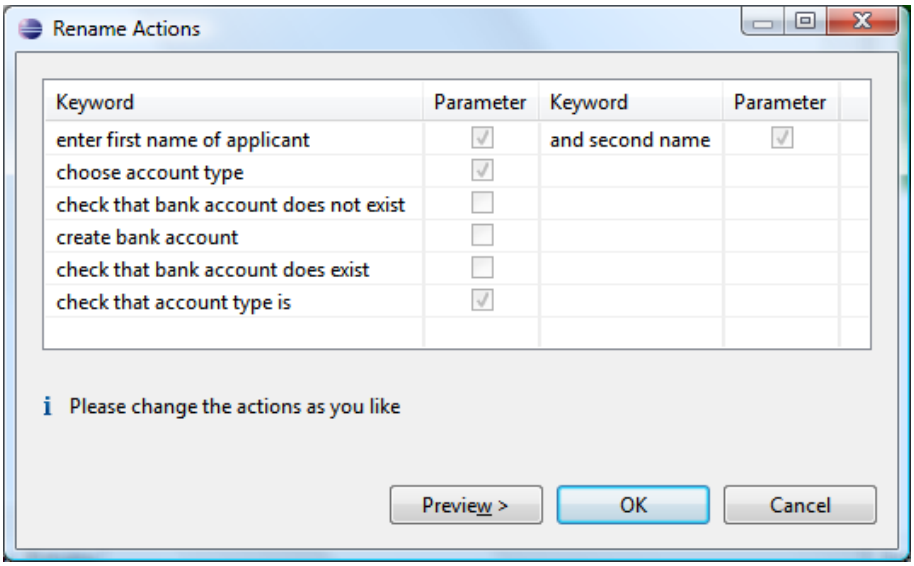

Figure 6.30: Input User Interface Mask of "Rename Action" Refactoring

The values of the keyword cells in the table can be modified after double-clicking on it. It is possible to modify as many cells as required before continuing to the preview. In other words, the user renames actions by changing the keywords in the table. Every keyword of an action might have a parameter associated which is indicated by a checkbox after the keyword.

There are two constraints regarding valid input:

⇒ **New name valid Java identifier**

When the keywords of the actions are camel-cased they must be valid Java method identifier as this value is used for the new method name in the fixture. The validity gets checked with help of the JavaConventions.validateFieldName() and JavaConventions.validateMethodName() methods that are provided by the Eclipse framework.

⇒ **Unique action**

When the keyword of an action is changed so that the resulting action is a duplication of an existing action the wizard notifies the user about the invalid input by asking him to change the keywords to an unique name.

Implementation Specifics

The “Rename Action” refactoring has no implementation specifics.

6.7.5 Add Action

The “Add Action” refactoring implements the refactoring description given in 5.5.3.2. After selecting the refactoring for an acceptance test in the test storage view, the user sees a user interface mask (see low-fidelity prototype in Figure 6.31).

Figure 6.31: Low-Fidelity Prototype of "Add Action" Refactoring

The implementation of the low-fidelity prototype can be seen in Figure 6.32. The user interface mask consists of two tables and two buttons to modify the actions.

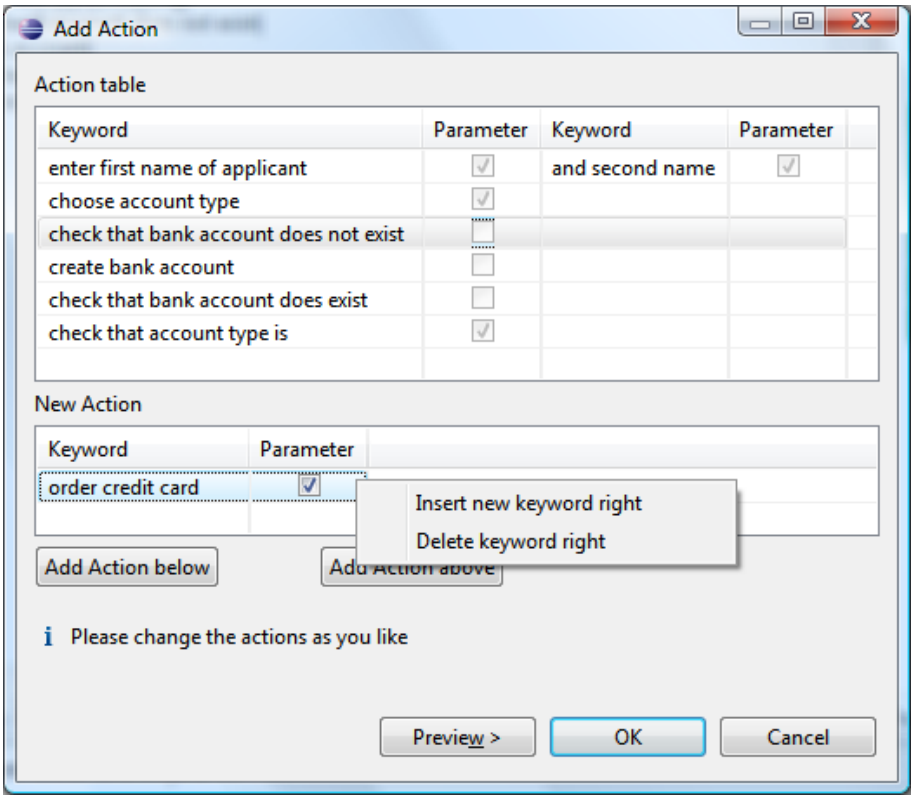

Figure 6.32: Input User Interface Mask of "Add Action" Refactoring

The upper table shows all existing actions similar to the user interface of the “Rename Action” refactoring task. The lower table is used to create a new action. The number of keywords can be adjusted by right clicking on the table and choosing a command from the context menu to either add or remove a keyword. The keywords can be edited and the checkbox indicates if a parameter follows the keyword. When the user has finished creating a new action, a position in the upper table can be chosen und the action will be inserted above or below the selected position by clicking on the respective buttons.

There is one constraints regarding valid input:

⇒ **New name valid Java identifier**

When the keywords of the actions are camel-cased they must be valid Java method identifier as this value is used for the new method name in the fixture. The validity gets checked with help of the JavaConventions.validateFieldName() and JavaConventions.validateMethodName() methods that are provided by the Eclipse framework.

Implementation Specifics

When an action is added and the respective method already exists in the fixture, the refactoring will be completed without adding a method.

6.7.6 Remove Action

The “Remove Action” refactoring implements the refactoring description given in 5.5.3.3. After selecting the refactoring for an acceptance test in the test storage view, the user sees a user interface mask (see low-fidelity prototype in Figure 6.33).

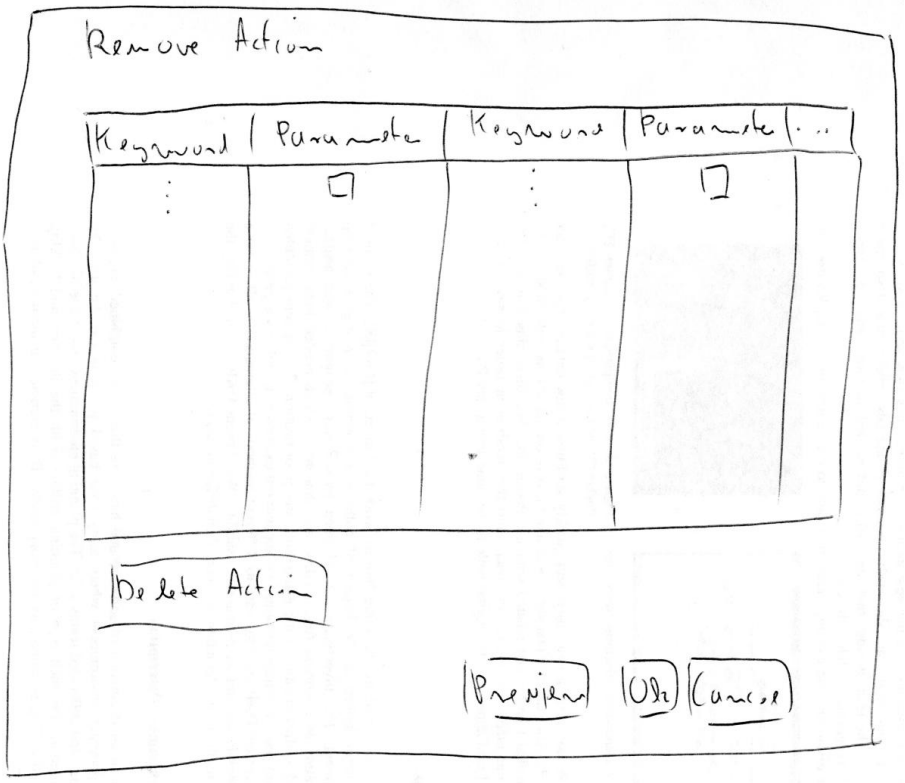

Figure 6.33: Low-Fidelity Prototype of "Remove Action” Refactoring

The implementation of the low-fidelity prototype can be seen in Figure 6.34. The user interface mask consists of one table and one button.

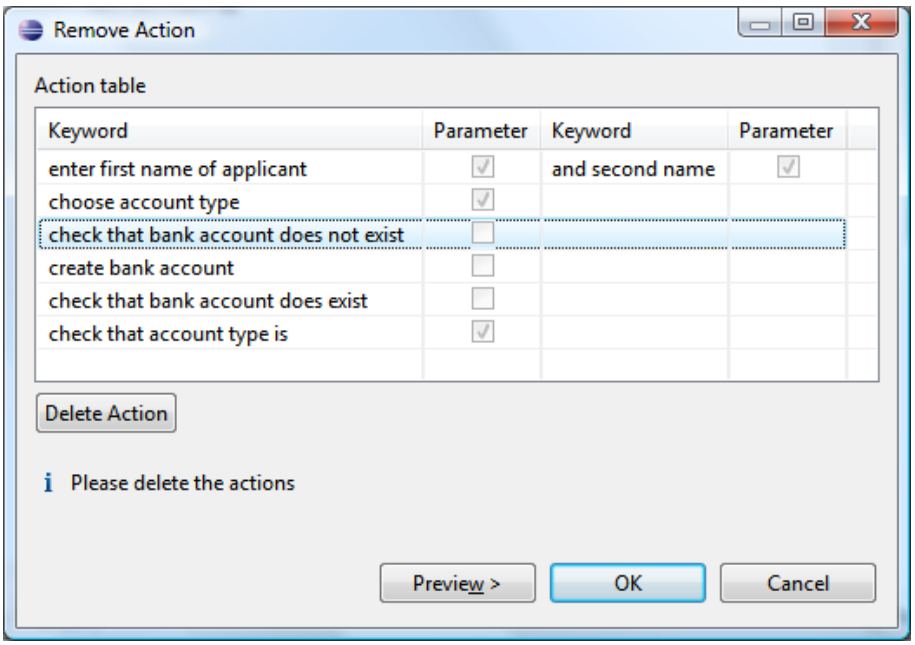

Figure 6.34: Input User Interface Mask of "Remove Action" Refactoring

The table shows all existing actions like the user interface of the “Rename Action” refactoring task. The user can select an action in the table and click on “Delete Action” which will remove the selected action from the table. It is also possible to remove multiple actions before proceeding to the preview.

Implementation Specifics

When an action is removed and the connected method does not exist in the fixture, the refactoring will be completed without removing the method.

6.8 Correctness of the System

The FitClipse refactoring extension has been built using Executable Acceptance Test Driven Development and therefore includes unit tests as well as acceptance tests. Every important component gets tested by unit tests and every refactoring task gets tested by an acceptance tests. However, due to limitations of the FIT framework which is not capable of executing fixtures in an Eclipse instance (for testing Eclipse plug-ins), a workaround has been applied.

PDE JUnit (Eclipse, 2004) runs JUnit tests in an separate Eclipse instance and is mainly used to test Eclipse plug-ins. The acceptance tests for the refactoring tasks are written as workflows using the DoFixture. As shown in 5.4, every action is mapped to a specific method in the fixture code.

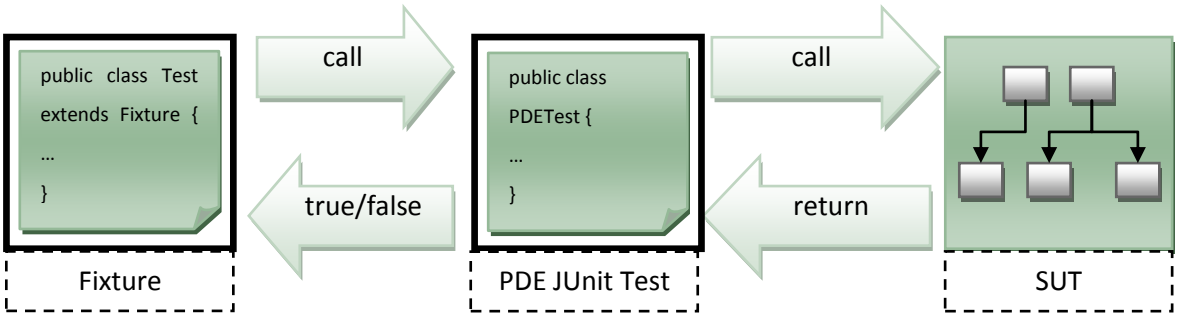

Figure 6.35: Refactoring Acceptance Testing Workaround

These specific methods open an Eclipse instance and call methods in the PDE JUnit test which is running in that instance. The PDE JUnit test method is doing the actual testing in the system and returns a Boolean value which is processed by the fixture to indicate whether the action was successful or not.

This way of executing the acceptance tests is very slow because an Eclipse instance has to be opened for every action in the fixture. A single test might need several minutes before it is completed.

6.9 Limitations

The FIT framework supports multiple test tables in one test definition of different types. In other words, an acceptance test can contain a ColumnFixture table and a DoFixture table together in one test or test definition. Due to time constraints, this mixed test definitions have not been implemented. The FitClipse refactoring tool supports multiple test tables in one test definition but tables have to be of the same type and structure. However, this has no impact on the performed analysis but is a limitation in the flexibility of the refactoring.

7 Conclusion and Future Work

7.1 Problems

The goal of this thesis was to find ways to make refactoring applicable for acceptance tests and to extend FitClipse to support automated refactoring.

An early research about acceptance test refactoring showed that not much work in this area has been done in the past. A few publications by Jennitta Andrea were available that discussed the need for acceptance test refactoring but not how it should be performed. Due to time constraints, it was not possible to conduct a survey to find out what types of refactoring would be beneficial for experienced users in the industry. Therefore, a set of refactoring types has been created but, unfortunately, there is no evidence that it is useful.

The next problem was to gain a proper understanding of the Eclipse framework. It was very time consuming to find out how to work with the Language Toolkit and Java Development Tools plug-ins for the refactoring integration. There is no literature available that describes the Eclipse refactoring framework and the available information on the Internet is also rather poor. In the end, a small tutorial describing a simple refactoring (Frenzel, 2006), a more sophisticated description of “Introduce Indirection” refactoring for Java (Widmer, 2007) and a tutorial about the Abstract Syntax Tree (Kuhn, et al., 2006) helped to get started.

Since the refactoring extension has been developed by using Executable Acceptance Test Driven Development the FIT framework was used to create executable acceptance tests which test the refactoring support. After finishing the first refactoring and implementing the fixture code of the corresponding acceptance test, the problem that the FIT framework was not capable of executing and testing Eclipse plug-ins came up. A quick research showed that there was no official solution and thus a small extension had to be built to make it work. Although it is working and the refactoring tasks can be tested by acceptance tests, it is extremely slow and not very well designed.

7.2 Contributions

The work presented in this thesis can be seen as a first step towards a better understanding of acceptance test refactoring. The following contributions have been made by this work:

⇒ **Academic contribution**

As a foundation for the analysis of acceptance tests and possible refactoring types, a definition of acceptance test refactoring has been created. It defines exactly the goal and differentiates between behaviour changing and behaviour preserving refactoring tasks. Furthermore, based on the definition a catalogue of acceptance test refactoring for ColumnFixture and DoFixture has been created. This academic work can be used as a base for additional research on this topic to get a deeper and more comprehensive understanding of acceptance test refactoring.

⇒ **Refactoring support of FitClipse**

Besides the academic contributions, an automated refactoring tool support has been developed. FitClipse has been extended to allow users to refactor acceptance tests by type of DoFixture and ColumnFixture. A total of six refactoring tasks were developed to support users changing acceptance tests. The test definition and fixture are kept consistent and in the case of a behaviour change, the user gets a clear message explaining the problem.

⇒ **Integration of Eclipse refactoring framework and extendibility**

Not only has FitClipse been extended to support refactoring, but also a set of interfaces have been created that allows an easy integration of new fixtures and refactoring types into FitClipse. New fixtures can be implemented by simply realizing one interface. The fixture parser as well as the test definition parser support many generic operations needed to add new refactoring types.

7.3 Future Work

Several problems and limitations have to be solved in the future:

⇒ **Evaluation**

The research of this work has not included an evaluation of the usefulness and usability (except using the tool to maintain its own acceptance tests). The next step is to determine if the refactoring support is beneficial to making the EATDD process easier and safer with respect to changing requirements.

⇒ **Extended fixture support**

Currently DoFixture and ColumnFixture are supported. These two fixtures are very popular and used very often (based on informal discussions with industry contacts). However, many additional fixtures like DomainFixture (for Domain Driven Design) or RowFixture (for checking results of queries) should be supported to make FitClipse attractive to a larger user group. This makes it necessary to analyse the relationships between the appropriate fixture and the test definition (as done in this work).

⇒ **Extending set of refactoring tasks**

To give the user more flexibility in modifying acceptance tests, the catalogue of refactoring tasks must be extended to support more kinds of refactoring. Furthermore, existing refactoring types could be combined to perform several changes in one-step in order to build even more sophisticated kinds of refactoring.

⇒ **Multiple test tables**

Currently, FitClipse allows the refactoring of multiple test tables in one test definition. Nevertheless, there is the limitation that every table must be of the same test type (e.g. DoFixture) and must have the same structure. Due to this fact, the user is forced to separate test tables into different acceptance tests which can lead to a higher maintenance effort.

⇒ **Support for more test definition formats**

The current implementation works with wiki syntax for writing the test definition. This notation is not very easy to use for users who have never worked with FitNesse before. Since the FIT framework supports HTML as input format, a HTML editor in FitClipse would be beneficial for non-technical users to create the appropriate tables or tests. The refactoring support would have to be extended to support HTML. This can easily be achieved by extending the ITestDefinition interface and implementing a parser that is able to work with HTML tables instead of wiki tables.

⇒ **Refactoring history**

The Eclipse framework supports keeping a history of applied refactoring changes to an element like a Java file or an acceptance test. That feature can be used to track all changes to acceptance tests, which may be helpful for developers and customers to detect requirements that have been altered. Furthermore, it builds a safety net that can be used to undo changes to acceptance tests if the new test does not match the expectations.

⇒ **Refactoring from Java file to acceptance test**

The actual refactoring implementation only allows users to perform a one-way refactoring from acceptance test to fixtures. Additionally, it would be beneficial to refactor from the Java file to the acceptance test. For example, a user removes a field in a ColumnFixture and the appropriate column in the test definition is automatically removed. This would offer developers to improve the structure or design of the fixture code without the need to keep the test definition consistent.

⇒ **Acceptance testing of refactoring support**

As shown in the problems section of this chapter, acceptance tests to check the refactoring support of FitClipse have been created. However, the solution built for the FIT framework to test the Eclipse plug-in is very slow and not well designed. This makes automated testing almost impossible due to the time a test run takes. Another way to execute the test faster has to be discovered and it should be considered optimize the design. This would make it possible to test the refactoring extension with a continuous integration server automatically. Additionally, the acceptance test's regression safety net could be used better for future development.

8 References

Aarniala, Jari. 2006. Acceptance Testing. [Online] 2006. [Cited: March 25, 2008.] http://www.cs.helsinki.fi/u/jaarnial/jaarnial-testing.pdf.

Ambler, Scott W. 2007. Introduction to Test Driven Design (TDD). [Online] 2007. [Cited: March 25, 2008.] http://www.agiledata.org/essays/tdd.html.

Andrea, Jennitta. 2005. Brushing Up On Functional Test Effectiveness. [Online] 2005. [Cited: March 10, 2008.] http://www.jennittaandrea.com/wp-content/uploads/2007/03/brushinguponfunctionaltesteffectiveness_bettersoftware2005.pdf.

—. 2005. Brushing Up On Functional Test Effectiveness. [Online] 2005. [Cited: March 10, 2008.] http://www.jennittaandrea.com/wp-content/uploads/2007/03/brushinguponfunctionaltesteffectiveness_bettersoftware2005.pdf.

—. Brushing Up On Functional Test Effectiveness. [Online] [Cited: March 10, 2008.] http://www.jennittaandrea.com/wp-content/uploads/2007/03/brushinguponfunctionaltesteffectiveness_bettersoftware2005.pdf.

—. 2007. Envisioning the Next Generation of Functional Testing Tools. *IEEE Computer Society.* May 2007, Vol. 24, 3, pp. 58-66.

ASE Group. 2008. EATDD. [Online] University of Calgary, 2008. [Cited: March 25, 2008.] http://ase.cpsc.ucalgary.ca/ase/index.php/EATDD/Home.

—. 2008. FitClipse. [Online] University of Calgary, 2008. [Cited: March 12, 2008.] http://ase.cpsc.ucalgary.ca/index.php/FitClipse/FitClipse.

Astels, David. 2003. *Test-Driven Development: A Practical Guide.* s.l. : Prentice Hall, 2003.

AutAT. 2005. [Online] 2005. [Cited: March 11, 2008.] http://boss.bekk.no/boss/autat.

Beck, Kent and Andres, Cynthia. 2004. *Extreme Programming Explained: Embrace Change.* s.l. : Pearson Education, 2004.

Beck, Kent. 1999. *Extreme Programming Explained: Embrace Change.* s.l. : Addison Wesley, 1999.

—. 2003. *Test-Driven Development: By Example.* s.l. : Addison-Wesley, 2003.

Binder, Robert. 2000. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* s.l. : Addison-Wesley, 2000.

Buwalda, Hans. 2004. Soap Opera Testing. *Better Software Magazine.* 2004.

Cockburn, Alistair. 2004. *Crystal Clear: A Human-Powered Methodology for Small Teams.* s.l. : Addison-Wesley Professional, 2004.

Cohn, Mike. 2005. Do-It-Yourself: A How-to Guide for Fixing a Failing Project. *Better Software Magazine.* 2005, Vol. 7, 8.

—. 2004. *User Stories Applied: For Agile Software Development.* s.l. : Pearson Education, 2004.

ConFIT. 2007. [Online] 2007. [Cited: March 11, 2008.] http://bandxi.com/fitnesse/confit.html.

Craig, Rick D. and Jaskiel, Stefan P. 2002. *Systematic Software Testing.* s.l. : Artech House, 2002.

Crispin, Lisa and House, Tip. 2002. *Testing: Extreme Programming.* s.l. : Pearson, 2002.

Cunningham, Ward. Fit: Framework for Integrated Test. [Online] http://fit.c2.com/.

Deng, Chengyao, Wilson, Patrick and Maurer, Frank. 2007. FitClipse: A Fit-based Eclipse Plug-in For Executable Acceptance Test Driven Development. *Proceedings of the 8th International Conference on Agile Processes in Software Engineering and eXtreme Programming (XP 2007).* 2007.

Deursen, Arie van and Moonen, Leon. 2002. The Video Store Revisited – Thoughts on Refactoring and Testing. *Proceedings of the 3rd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP 2002).* 2002.

Deursen, Arie van, et al. 2001. Refactoring Test Code. *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001).* 2001.

Eclipse Foundation. 2008. JDT. [Online] 2008. [Cited: March 21, 2008.] http://wiki.eclipse.org/JDT.

—. 2004. What is a PDE JUnit test? [Online] 2004. [Cited: March 17, 2008.] http://wiki.eclipse.org/FAQ_What_is_a_PDE_JUnit_test.

—. 2007. What is LTK? [Online] 2007. [Cited: March 16, 2008.] http://wiki.eclipse.org/FAQ_What_is_LTK.

Erickson, Carl, et al. 2003. Make Haste, not Waste: Automated System Testing. *Extreme Programming and Agile Methods - XP Agile Universe, Springer Lecture Notes in Computer Science.* 2003.

FitLibrary. [Online] [Cited: March 25, 2008.] http://sourceforge.net/projects/fitlibrary.

FitNesse. 2008. [Online] 2008. [Cited: March 11, 2008.] http://fitnesse.org.

—. 2008. [Online] 2008. [Cited: March 11, 2008.] http://fitnesse.org.

Fowler, Martin. 2000. *Refactoring: Improving the Design of Existing Code.* s.l. : Addison-Wesley, 2000.

—. 2006. Specification By Example. [Online] 2006. [Cited: March 25, 2008.] http://martinfowler.com/bliki/SpecificationByExample.html.

Framework for Integrated Test. [Online] [Cited: March 25, 2008.] http://fit.c2.com/.

Frenzel, Leif. 2006. The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs. [Online] 2006. [Cited: March 15, 2008.] http://www.eclipse.org/articles/Article-LTK/ltk.html.

Gamma, Erich and Beck, Kent. 2003. *Contributing to Eclipse: Principles, Patterns, and Plug-Ins.* s.l. : Addison-Wesley, 2003.

Gamma, Erich. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* s.l. : Addison-Wesley, 1995.

Guerra, Eduardo Martins and Fernandes, Clovis Torres. 2007. Refactoring Test Code Safely. *International Conference on Software Engineering Advances (ICSEA 2007), Cap Esterel, France.* 2007.

Highsmith, James A. 2000. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems.* s.l. : Dorset House Pub., 2000.

IEEE. 1996. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. 1996.

Javvin. 2007. *Network Dictionary.* s.l. : Javvin Technologies Inc, 2007.

Jeffries, Ron. 2001. What is Extreme Programming? [Online] 2001. [Cited: March 25, 2008.] http://www.xprogramming.com/xpmag/whatisXP.htm.

Kaner, Cem. 2003. Cem Kaner on Scenario Testing: The Power of What-If and And Nine Ways to Fuel your Imagination. *Software Testing and Quality Engineering Magazine.* 2003, Vol. 5, 5.

—. 2003. How to design scenario tests. *Software Testing and Quality Engineering Magazine.* 2003.

Kaner, Cem, Bach, James and Pettichord, Bret. 2002. *Lessons Learned in Software Testing: A ContextDriven Approach: A Context Driven Approach.* s.l. : John Wiley & Sons, 2002.

Kastens, Uwe, Waite, William McCastline and Sloane, Anthony M. 2007. *Generating Software from Specifications.* s.l. : Jones & Bartlett Publishers, 2007.

Kerievsky, Joshua. 2005. Industrial XP: Making XP Work In Large Organizations. *Cutter Consortium: Agile Project Management.* 2005, Vol. 6, 2.

—. 2004. Storytesting. [Online] 2004. [Cited: March 13, 2008.] http://www.industrialxp.org/storytesting.html.

Kuhn, Thomas and Thomann, Olivier. 2006. [Online] 2006. [Cited: March 16, 2008.] http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html.

Luxoft. 2007. FITPro. [Online] 2007. [Cited: March 11, 2008.] http://www.luxoft.com/fit.

Maciaszek, Leszek A. 2001. *Requirements Analysis and System Design.* s.l. : Addison-Wesley, 2001.

Manifesto for Agile Software Development. 2001. [Online] 2001. [Cited: March 25, 2008.] http://agilemanifesto.org.

Marick, Brian. 2003. Agile Testing Directions: Business-facing team support. [Online] 2003. [Cited: 03 13, 2008.] http://www.testing.com/cgi-bin/blog/2003/09/05#agile-testing-project-4.

—. 2003. Exploration Through Example. [Online] 2003. [Cited: March 25, 2008.] http://www.testing.com/cgi-bin/blog/2003/08/21.

—. 2002. XP/Agile Universe trip report. [Online] 2002. [Cited: March 25, 2008.] http://www.pettichord.com/XP_Agile_Universe_trip_report.txt.

Mathew, Sajan. 2003. *Software Engineering.* s.l. : S. Chand & Company, 2003.

Maurer, Frank and Melnik, Grigori. 2006. Driving Software Development with Executable Acceptance Tests. *Agile Project Management.* 2006, Vol. 7, 11.

—. 2007. Multiple Perspectives on Executable Acceptance Test-Driven Development. *Proceedings of the 8th International Conference on Agile Processes in Software Engineering and eXtreme Programming (XP 2007).* 2007.

Melnik, Grigori. 2007. Empirical Analysis of Executable Acceptance Test Driven Development. *Agile Software Engineering Group.* [Online] August 2007. [Cited: March 25, 2008.] http://ase.cpsc.ucalgary.ca/ase/uploads/Publications/MelnikPhD.pdf.

Meyers, Glenford J. 2004. *The Art of Software Testing.* s.l. : John Wiley and Sons, 2004.

Palmer, Stephen R. and Felsing, John M. 2002. *A Practical Guide to Feature-Driven Development.* s.l. : Prentice Hall, 2002.

Park, Shelly and Maurer, Frank. 2008. Multi-modal Functional Test Execution. *Proceedings 9th International Conference on Agile Processes and eXtreme Programming in Software Engineering (XP2008), Limerick, Ireland, Springer, 10-14 June 2008.* 2008.

Pyxis Technologies Inc. 2008. GreenPepperSoftware. [Online] 2008. [Cited: March 11, 2008.] http://greenpeppersoftware.com/en/products.

Schwaber, Ken and Beedle, Mike. 2002. *Agile Software Development with SCRUM.* s.l. : Prentice Hall, 2002.

Stapleton, Jennifer. 1997. *Dynamic Systems Development Method: The Method in Practice.* s.l. : Addison-Wesley, 1997.

The Standish Group. 1995-2005. *Chaos Report.* s.l. : The Standish Group, 1995-2005.

US Department of Defense. 1988. Military Standard Defense System Software, DOD-STD-2167A. *Section 5.3.3. Formal Qualification Testing.* [Online] 1988. [Cited: March 25, 2008.] http://www2.umassd.edu/swpi/dod/mil-std-2167a/dod2167a.html#section.5.3.

Wang, Lingfeng and Chen Tan, Kay. 2006. Modern Industrial Automation Software Design. s.l. : Wiley-IEEE, 2006.

Watkins, John. 2001. *Testing IT: An Off-the-Shelf Software Testing Process.* s.l. : Cambridge University Press, 2001.

Widmer, Tobias. 2007. Unleashing the Power of Refactoring. [Online] 2007. [Cited: March 15, 2008.] http://www.eclipse.org/articles/article.php?file=Article-Unleashing-the-Power-of-Refactoring/index.html.

Young, Ralph Rowland. 2001. *Effective Requirements Practices.* s.l. : Addison-Wesley, 2001.