Refactoring of Acceptance Tests in Visual Studio

Denis Elbert

Diplomarbeit

Studiengang Informatik (Technik)

Fakultät für Informatik
Hochschule Mannheim

Autor:	Denis Elbert Matrikelnummer: 510243
Zeitraum:	16.11.2009 – 16.03.2010
Erstgutachterin:	Prof. Dr. Astrid Schmücker-Schend
Zweitgutachterin	Prof. Dr. Miriam Föller-Nord
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

Statutory Declaration (German)

Ich versichere, dass ich die vorliegende Arbeit selbstständig 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, 16.03.2010 _____

Unterschrift

Abstract

Executable Acceptance Test Driven Development (EATDD) is an extension of *Test Driven Development* (TDD). TDD requires that unit tests are written before any code. EATDD pushes this TDD paradigm to the customer level by using *Acceptance Tests* to specify the requirements and features of a system. The *Acceptance Tests* are mapped to a *Fixture* that permits the automated execution of the tests.

With ongoing development the requirements of the system can change. Thus, the *Acceptance Tests* must be adjusted in order to reflect the new requirements. Since the tests and the corresponding *Fixtures* must remain consistent, the manual modification of these tests is time consuming and error-prone. Hence comes the need for *Acceptance Test* refactoring.

This thesis describes the implementation of *Acceptance Test* refactoring support for the Visual Studio IDE.

German Abstract

Executable Acceptance Test Driven Development (EATDD) ist eine Erweiterung des Test Driven Development (TDD). TDD schreibt vor, dass das Schreiben von Code erst nach Definition eines dafür vorgesehen Modultests durchgeführt wird. EATDD wendet dieses Prinzip bis auf Kundenebene an, indem Acceptance Tests verwendet werden, die die Anforderungen und Leistungsmerkmale des zu entwickelten Systems definieren. Diese Acceptance Tests sind mit einem *Fixture* verknüpft, das die automatisierte Ausführung dieser Tests ermöglicht.

Mit fortschreitender Entwicklung können sich die Anforderungen des Systems ändern. Als Folge dessen müssen die Acceptance Tests angepasst werden, um den geänderten Anforderungen Rechnung zu tragen. Da die Tests gegenüber den zugehörigen *Fixtures* konsistent bleiben müssen, ist eine manuelle Anpassung der Tests zeitaufwendig und fehleranfällig. Daraus resultiert die Notwendigkeit von Acceptance Test Refactoring.

Diese Arbeit beschreibt die Erweiterung von Visual Studio um Acceptance Test Refactoring - Funktionen.

ACKNOWLEDGEMENTS

First of all I would like to thank everyone, who supported me during this work and especially in the course of my whole studies.

In particular I would like to thank these people:

- Prof. Dr. Schmücker Schend for supervising my thesis.
- Prof. Dr. Maurer for giving me the opportunity to be part of his great team in Calgary.
- All members of the ASE group, who always helped me out when I got questions; especially Felix, Sabine and Steffen.
- My friend Sebastian for his great support

DEDICATION

I dedicate this work to my parents who always supported me whenever they could and let me choose my own way in life.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

CIL	Common Intermediate Language
CLR	Common Language Runtime
DTE	Development Tools Extensibility
EATDD	Executable Acceptance Test Driven Development
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
JITter	Just-In-Time Compiler
MSDN	Microsoft Developer Network
RDM	Requirements Definition Management
SDK	Software Development Kit
SUD	System Under Development
SUT	System Under Test
TDD	Test Driven Development
VS	Visual Studio
VSCT	Visual Studio Command Table
WPF	Windows Presentation Foundation
XAML	eXtensible Application Markup Language
XML	Extensible Markup Language
XP	Extreme Programming

1 Introduction

1.1 Motivation

During the last years, *Agile Software Development* gained more and more recognition as it is a qualified alternative to the traditional heavyweight software development processes. It focuses on team development and communication between the team members as well as between the customer and the developers.

Extreme Programming (XP) is one of the most agile software development processes [Astels 2003]. Its major practice is the application of *Test Driven Development* (TDD), which requires to write a test before any code is written. As part of XP, story cards are used to specify the requirements of a system from the customer's perspective. These story cards are translated into *Acceptance Tests* which represent the specification of the system.

In order to facilitate the creation of a safety net of tests, these *Acceptance Tests* must be executed in an automated process. The Automation requires the definition of a so called *Fixture,* which is used to intermediate between the test specification and the *System Under Test* (SUT).

Since requirements change with time, it is necessary to adjust the *Acceptance Tests* so they reflect exactly the customer's needs. Refactoring of *Acceptance Tests* supports this process of test modification because it keeps test specification and *Fixture* consistent. So far, *Acceptance Tests* refactoring support is only available for the Java environment with the *FitClipse* tool. Although *Acceptance Tests* frameworks such as *GreenPepper* exist for .NET, there is no tool yet which incorporates *Acceptance Test* refactoring in the development process of .NET applications.

1.2 Thesis Goals

The goal of this thesis is to implement refactoring functionality for *GreenPepper* acceptance tests in Visual Studio.

GreenPepper specifies different types of acceptance tests. Among them are the *RuleFor* and

the *Scenario* test (see chapter 3.5). Refactoring shall be supported only for these two acceptance test types. All refactoring actions that must be implemented in the course of this work are listed below:

- Rename test (both *Scenario* and *RuleFor* test)
- Add given / expected parameter column to *RuleFor* test
- Delete given / expected parameter column from *RuleFor* test
- Rename given / expected parameter column of *RuleFor* test
- Add action to *Scenario* test
- Delete action from *Scenario* test
- Edit / Rename action of *Scenario* test

There are some restrictions or general conditions that apply to these goals:

- The refactoring support must be implemented for the newest version of the .NET framework (version 4.0) and Visual Studio (version 2010).
- The programming language C# must be supported by the refactoring.

1.3 Thesis Structure

This thesis is organized in five major remaining parts.

Chapter 2 gives information about two projects which are related to the topic of this thesis.

It is followed by chapter 3, which explains *Agile Methods*, *Extreme Programming* (XP) and *Test Driven Development* as part of XP in order to clarify the relevance of acceptance test refactoring. In the same chapter, all fundamentals of *GreenPepper* acceptance tests are explained as well as any technology that is used in the course of this thesis.

Chapter 4 explains the approach that was taken in order to achieve the goals defined above, whereas chapter 5 provides detailed information about the final results and the actual

implementation.

The last chapter summarizes the entire work and describes possible future work.

2 Related Work

2.1 FitClipse

FitClipse is an Eclipse plug-in for facilitating *Executable Acceptance Test Driven Development* (EATDD) (see chapter 3.3.3) and was developed at the University of Calgary. The tool allows users to "edit acceptance tests, automatically generate fixtures, execute tests and represent the tests graphically including an option to view the test results history [Maurer et al. 2009a]". It supports both *Fit* and *GreenPepper,* which represent frameworks for acceptance tests.

2.2 GreenPepe 2010

GreenPepe 2010 is an extension for the integrated development environment (IDE) *Visual Studio 2010* and was developed by Felix Riegger and Denis Elbert at the University of Calgary in September 2009.

The extension makes it possible to execute *GreenPepper* acceptance tests (see chapter 3.5) within the IDE and to view the test results in a JUnit-like manner. The detailed features of *GreenPepe 2010* are listed and described below:

- **Marking files as GreenPepper acceptance tests**
  GreenPepper acceptance tests are specified within common HTML files. Since a project may not only contain HTML files that represent *GreenPepper* acceptance tests, *GreenPepe 2010* allows for marking those in order to distinguish them from other HTML files. When it comes to the execution of acceptance tests, *GreenPepe 2010* will skip all files which are not marked as *GreenPepper* acceptance tests.

- **Execution of multiple GreenPepper acceptance tests**
  GreenPepe 2010 offers the possibility to simply select *GreenPepper* acceptance tests through the context menu of the Visual Studio solution explorer and to execute them at once. If a folder or the root entry of the entire project is selected, *GreenPepe 2010* will find all containing *GreenPepper* acceptance test files and execute them.

- **Graphical overview over test results**

After *GreenPepper* acceptance tests have been executed, an additional window is displayed which gives an overview over all test results (number of successful or failed tests, exceptions, ignored tests). Furthermore, each executed test is listed and coloured depending on its result. A green colour represents a passed test while a red colour stands for a test that failed.

- **Display of GreenPepper acceptance test output**

  When a double click is performed on one of the tests listed in the test result view mentioned above, the associated test result (output from the *GreenPepper* engine, see chapter 3.5.3) is automatically opened and displayed in the IDE.

- **Rerun tests**

  In the test result window, it is also possible to select one or more acceptance tests in order to execute them again. This is useful when changes are applied to the system and the tests need to be run multiple times to check if they pass.

Figure 1 shows a screenshot of *GreenPepe 2010* taken after a number of acceptance tests have been executed. The test result window can be seen on the right side. A red bar indicates that at least one test failed whereas coloured circles in front of each listed test give information about their exclusive result. Through double click on the “Division Test”, the test result output produced by the *GreenPepper* engine is displayed on the left side.

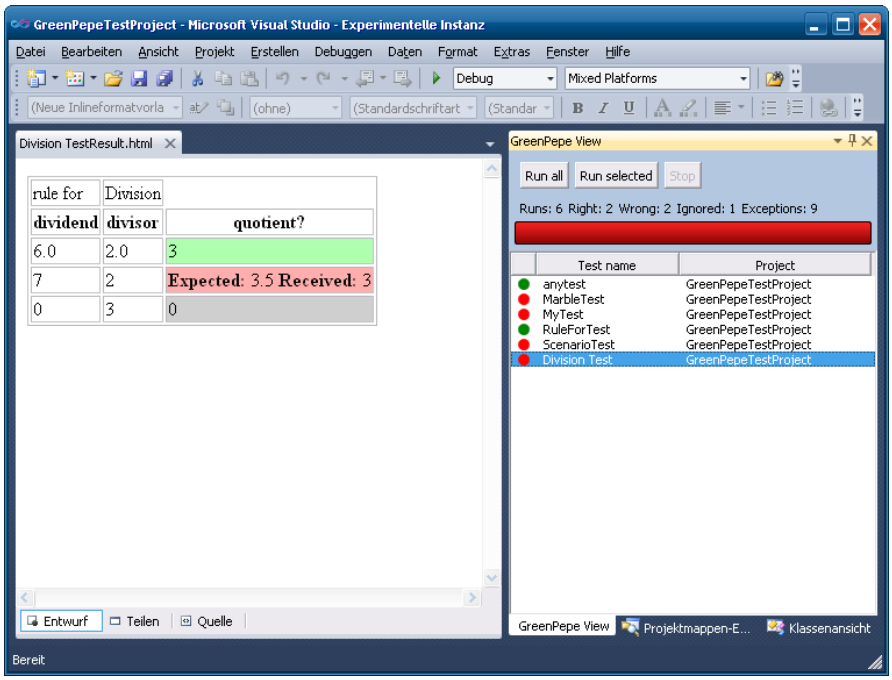


Figure 1: Screenshot of the GreenPepe 2010 extension

GreenPepe 2010 is implemented as a *VSPackage*. *VSPackages* represent one of three possible ways to extend Visual Studio (besides Macros and Add-Ins) [MSDN 2010b]. In order to execute *GreenPepper* acceptance tests, *GreenPepe 2010* references a library from the *GreenPepper* framework.

3 Fundamentals

3.1 Agile Software Development And Agile Methods

On February 11-13, 2001, seventeen representatives of various software development methodologies (such as Extreme Programming (XP), SCRUM, DSDM, Adaptive Software Development, Crystal, Feature-Driven Development, Pragmatic Programming) convened in the mountains of Utah to discuss alternatives to the traditional heavyweight methodologies. The result of their meeting was a draft of the *Agile Manifesto*, which defines the criteria for agile software development processes in four fundamental values and twelve principles.

The four values are stated as follows [Manifesto]:

- **Individuals and interactions** over processes and tools:
  Like the old adage "A fool with a tool is still a fool" implies, the most important factors to consider are the people writing the software and how they work together. The best processes and tools are not of any value as long as people are not able to communicate with each other.

- **Working software** over comprehensive documentation:
  When it comes to maintaining software, documents are very helpful as they give a detailed description of the final system. However, since the primary goal of software development is to create software, more time and effort should be spent on frequently building working product releases, which can be demonstrated to the customer.

- **Customer collaboration** over contract negotiation:
  In order to build the right software, it is important to communicate with your customer and integrate him into the software development process because he is the only one who knows the requirements of the system. That is why it is more important to stay in touch with the customer and respond to his requests than insist on the contract that was negotiated before.

- **Responding to change** over following a plan:
  Having a project plan is indispensable for a successful software development project,

> especially for large projects. But requirements change over time, which makes it necessary to adapt to those changes and redefine the project plan as soon as possible in order to satisfy the customer's wishes.

Each of the four statements consists of a left (emphasized) and a right side, which is explained by the authors of the *Manifesto* by saying: "[...] while there is value in the items on the right, we value the items on the left more [Manifesto]". In other words, the *Manifesto* does not question the importance of the traditional values on the right, but defines other values (on the left) that are considered to be even more important. It therefore does not specify alternatives but defines preferences [Ambler 2009a].

All software development processes that follow the idea of the *Manifesto* are known as *Agile Methods*. Examples for Agile Methods are: Extreme Programming (XP), SCRUM, Feature-Driven Development, Crystal, Dynamic Systems Development, Adaptive Software Development [Ordelt 2008, p.8].

Chapter 3.2 will provide detailed information about XP because this agile software development process leads to the application of Executable Acceptance Test Driven Development (EATDD) (see chapter 3.3.3) and refactoring of acceptance tests.

3.2 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., p. 2]" and is "one of the most agile of the agile processes [Astels 2003]". The foundation pillars of XP are the definition of values, practices and principles.

Practices, values and principles

Practices are concrete techniques or a set of repeatable actions. The technique of always writing a test before changing the code is a good example for a practice. Practices are clear and objective and define the way in which tasks should be carried out.

Values are the ideals a team or group agrees on in order to achieve an objective and represent what people like or dislike. While for practices it can be easily decided whether or not they

were respected properly, it is not possible for values in the same way. Considering communication as a good example for a value or ideal of a software development process, it can hardly be said that a person not attending a meeting does not value communication. The practice of attending the meeting, however, is obviously not respected by this person. Furthermore, practices are evidence of values because they produce effects that support them. In this example, regular meetings would support communication between the team members of the project [Power 2006].

The way in which values and practices are combined - that is to identify which practices have to be applied in order to achieve the goals and support the values - is called a *Principle.* Principles therefore are "[...] domain-specific guidelines for life [Beck et al., p. 15]" and bridge the gap between values and practices (see figure 2).

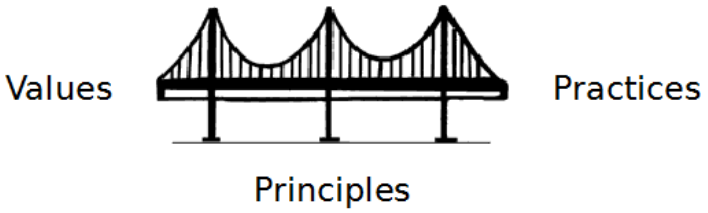


Figure 2: Correlation between Values, Principles and Practices [Beck et al., p. 15]

XP values

XP specifies a total of five values [Wells 2009]:

- **Communication**

  When working in a team, communication between all team members is indispensable. Communication helps to share knowledge gathered during the entire development process with all team members and improves the efficiency of the whole team.

- **Simplicity**

  XP achieves maximum value by focusing on what is needed and does not implement features that are not required.

- **Feedback**

  Every software development process has to face many changes in requirements,

design and architecture over the time. This makes it necessary to react as soon as possible to those changes in order to minimize the consequent overhead. Feedback is a very good way to do so and should therefore happen as frequently as possible.

- **Courage**

  It requires courage from each team member to confront all problems that may rise during the software development process. Courage is expressed in multiple ways such as patience (e.g., when the reason for a bug needs to be found) and honesty (e.g., when mistakes are made).

- **Respect**

  A team can only be successful, if the members respect each other as emancipated human beings and also care about the project they are working on.

XP principles

As mentioned before, principles bridge the gap between values and practices. The following list describes a chosen subset of XP principles that have an influence on the variety of practices that are used in XP [Beck et al., chapter 5]:

- **Humanity**

  Software is written by people. Keeping that simple fact in mind, it is not only important for a software development process to satisfy the business needs, but also to respect the personal needs of each human team member.

- **Failure**

  Not everything that is done will succeed. There will always be failures, especially when trying different approaches to find a solution. Failures do not necessarily need to be something bad. On the contrary, they can also impart knowledge and improve later steps by avoiding doing the same mistakes again.

- **Flow**

  “The practices of XP are biased towards a continuous flow of activities rather than discrete phases [Beck et al., p. 30]”. A good example of a flow-oriented approach is the daily build of the software. It makes sure that the final product works correctly and

reduces the risk of huge defects, which occur more likely the more time is in between the current and the last software build.

- **Small steps**

  When applying changes in small steps, the resulting overhead is much smaller compared to a momentous change taken all at once. Defects can be found more easily and fixed much faster and more cost-effectively. The test-first programming practice (test driven development) supports the principle of applying changes in small steps and will be explained in greater detail in chapter 3.3.

XP practices

Both values and principles decide on the practices applied in XP. The following list describes some primary practices of XP:

- **Sit together as a team**

  The whole team should be located in one room, where all team members can communicate with each other easily. Since people also need some privacy, private rooms should be provided where they can back out for a while.

- **Informative workspace**

  The workspace should be organized in a way that everyone involved in the project can easily catch up the current progress. A good example to achieve this, is to put story cards on a designated area of a wall, which provide information about the work that has already been done as well as upcoming work units.

- **Pair programming**

  “Pair programming is a dialogue between two people simultaneously programming (and analyzing and designing and testing) and trying to program better [Beck et al., p. 42]”. The partners sit in front of one machine and alternate with typing and examining the source code.

- **User stories**

  User stories are short descriptions of “units of customer-visible functionality [Beck et al., p. 44]”. Besides a short prose or graphical description, a user story includes an

estimation about the development effort necessary to implement it. Usually, user stories are noted down on index cards and attached to a wall, where every team member can see them.

- **Continuous integration**

  A major goal of XP is to integrate changes as fast as possible. “The longer you wait to integrate, the more it costs and the more unpredictable the costs become [Beck et al., p. 49-50]”. XP therefore requires to automate the process of integrating and testing code changes as well as to notify the development team immediately upon resulting errors.

- **Test-first programming**

  Another basic concept in XP is to “[...] write a failing automated test before changing any code [Beck et al., p. 50]”. This approach helps to make sure that the system does what it is expected to do and reduces the error-proneness because the new code is already covered by a test that can be run automatically.

3.3 Test Driven Development (TDD)

Test-Driven Development (TDD) or *Unit Test-Driven Development* (UTDD) is one of the main design tools (practices) in XP (see chapter 3.2) and therefore a core part of this agile process. It is not just another way of testing software, but a new style of software development. The basic concepts are [Astels 2003, chapter 1]:

- **Tests are written first before any code:**

  Whenever new functionality has to be added to the system, the respective code is not written immediately. Instead, a test is written in advance which will test if the new functionality meets the requirements and works as expected. Upon completion of the test, the functionality itself is implemented, until the test passes.

- **Tests determine what code needs to be written:**

  The first concept explained above leads to the result that no more code is written unless a test fails or a new test is added (which also fails initially since it is not implemented yet). In other words, as soon as all tests pass, no more code is written.

- **No code goes into production unless it has associated tests:**

  Due to the approach of writing tests first before the actual functionality is implemented and writing only code required to make the tests pass, tests can be deemed as representatives of the system's features. By implication, it can be said that "a feature does not exist until there is a suite of tests to go with it [Astels 2003, p. 7]".

- **TDD creates an exhaustive test suite**

  Following the above mentioned concepts, TDD creates – in theory – an exhaustive test suite because code is always written in connection with a test.

The activity diagram in figure 3 shows the typical steps when following the TDD approach.

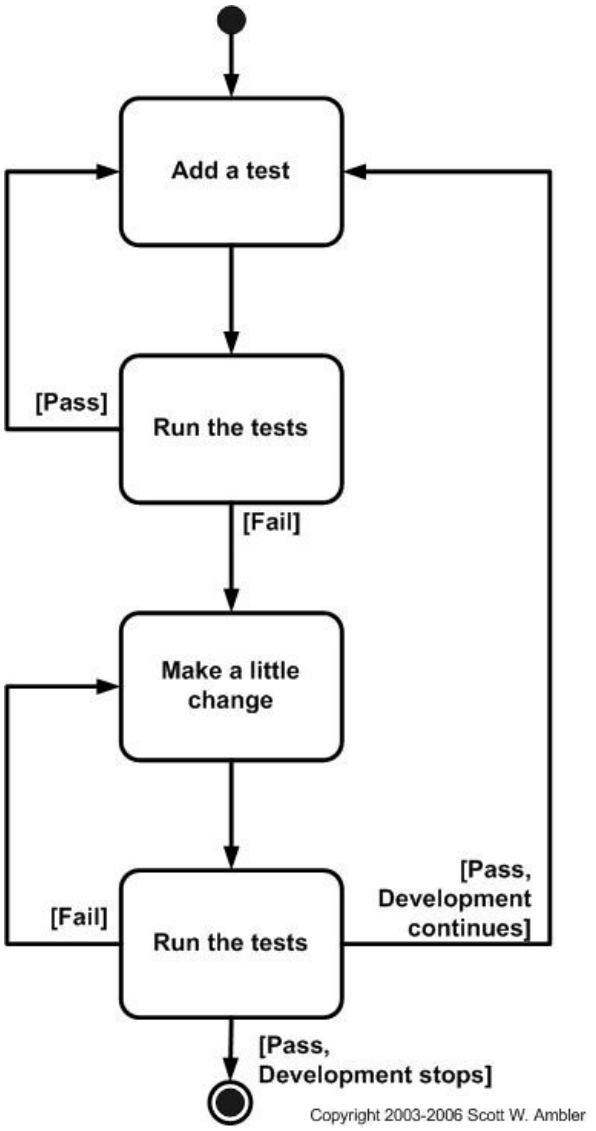


Figure 3: TDD steps [Ambler 2009b]

The steps are as follows:

(1) Add a test

A new test is created to specify and describe the functionality that has to be incorporated into the system.

(2) Run the tests

The newly added test is run for the first time and should fail because no code has been written yet to implement the new functionality.

(3) Make a little change

Code is implemented to make the newly added failing test pass.

(4) Run the tests

After applying the code changes, the test is run again to make sure that it passes. All other tests of the entire test suite are also run to make sure that the newly applied changes did not break already existing functionality. If one of the tests fails, it is started over with step 3. If all tests pass, the cycle can start from the beginning with adding a new test.

There are different types of tests that can be used along with TDD. In the traditional definition of TDD, so called *Unit Tests* are used, which are described in chapter 3.3.1. A higher abstraction of the system is achieved by using *Acceptance Tests* (see chapter 3.3.2). Applying TDD with *Acceptance Tests* is also known as *Executable Acceptance Test-Driven Development* (EATDD) and will be explained in chapter 3.3.3.

Regardless of which kind of tests are used to apply TDD, it requires to have a framework which allows for executing the tests automatically. Table 2 lists possible testing frameworks (list is not exhaustive) for each *Unit Tests* and *Acceptance Tests* that are available for the .NET or Java environment.

Kind of test	Platform / Programming language	Testing Framework
Unit test	.NET / C#	NUnit
Unit test	Java	JUnit
Acceptance test	.NET / C#	GreenPepper
Acceptance test	Java	FIT / GreenPepper

Table 1: TDD frameworks in different environments

As far as Microsoft's .NET environment is concerned, *GreenPepper* represents a framework to specify and execute *Acceptance Tests* and is explained in greater detail throughout this thesis (see also chapter 3.5).

3.3.1 Unit Tests

Unit Tests serve to test software on its lowest level of implementation from the developer's perspective. As their name implies, they are used to test the behaviour of a small unit of code and therefore test technical details of the system. In object oriented programming languages (like C# or Java), for instance, the smallest unit is represented by a method.

The goal of Unit Testing is to ensure that these units of an application are working as expected and meet the requirements. In the example above, each *Unit Test* is related to a method and verifies its functionality.

Unit Testing frameworks such as *NUnit* for the C# .NET programming language and *JUnit* for Java provide support for the execution of Unit Tests as well as the management of several test runs and test results [Koehler 2007, chapter 4.1].

3.3.2 Acceptance Tests

In contrast to *Unit Tests*, which test technical details of the system, *Acceptance Tests* are used to perform black box testing, i.e., they test the system as a whole from the customer's perspective. The objective of *Acceptance Testing* is to "[...] verify whether the functionalities of the system meet the requirements of the customer [Maurer et al. 2009]", whereas the motivation is to "[...] demonstrate working functionality rather than to find faults (although faults may be found as a result of acceptance testing) [Maurer et al. 2005]".

In *Extreme Programming* (XP), where *Acceptance Tests* are part of the TDD practice (see chapter 3.2), user stories are translated into *Acceptance Tests.* These tests are written by the customer and contain scenarios that test when the user story is considered to be correctly implemented and the system meets the customer's requirements [Wells 2009].

Since the manual execution of *Acceptance Tests* is very time consuming and error-prone, executable *Acceptance Tests* have been introduced, which allow for executing tests more frequently in an automated process. More details about executable *Acceptance Tests* can be found in chapter 3.3.3.

3.3.3 Executable Acceptance Test Driven Development

Executable Acceptance Test Driven Development (EATDD), also known as *Customer Test Driven Development* or *Story Test Driven Development,* is an extension of TDD (see chapter 3.3). It pushes the TDD paradigm to the customer level by using *Acceptance Tests* (instead of *Unit Tests*) to specify the requirements and features of a system. The purpose of EATDD is to improve the communication between the customer and the developers and "[...] to help developers better understand the requirements and validate their development with the customer's requirements [Maurer et al. 2007]".

In the world of *Agile Methods*, especially for *Extreme Programming* (XP) (see chapter 3.2), "[...] the iterative nature of the processes dictates automation of the acceptance tests (i.e. producing 'executable acceptance tests') as manual regression testing at the customer level is too time consuming to be practical and feasible given the short time frames of agile iterations [Maurer et al. 2007]".

In order to provide automation, EATDD mandates to write the requirements for a feature down in form of tests (*Executable Acceptance Tests*) rather than in natural language. Once the tests are written by the customer, the developers start to write the test code that tests whether or not the current state of the system, also called *System Under Test* (SUT) or *System Under Development* (SUD), meets the requirements [Maurer et al. 2008]. Since these tests can be executed, they are also called *Executable Specifications*.

Similar to TDD, but located on a higher abstraction level, EATDD requires "[...] that no code is written for a new feature unless an automated acceptance test fails. That means at least one customer acceptance test for a feature (also called story tests) needs to be developed before the development team starts tackling that feature [ASE 2009]". Figure 4 shows the typical EATDD workflow applied in XP.

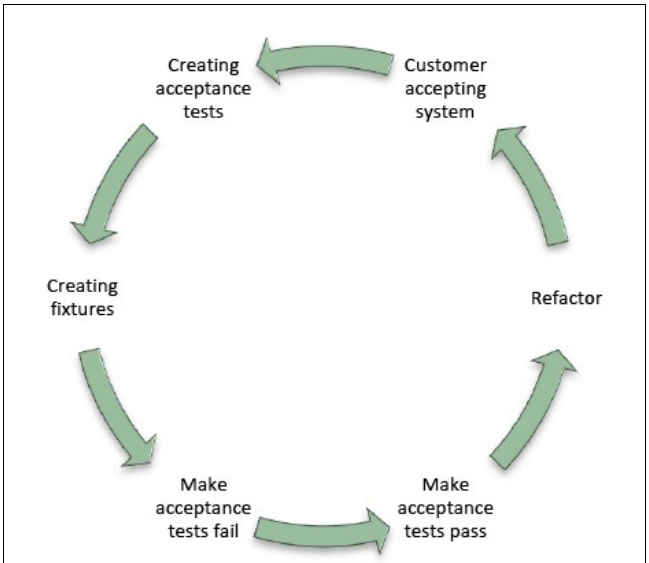


Figure 4: EATDD workflow in XP [Ordelt 2008, p. 18]

The steps are as follows [Ordelt 2008]:

(1) Creating Acceptance tests

Based on the *User Stories*, *Acceptance Tests* are created by customers and developers, which specify the requirements of a new feature.

(2) Creating fixtures

Fixtures represent the test code that is written by the developers to verify the SUT against the specification expressed by the *Acceptance Tests*. Based on these tests, the *Fixtures* are implemented by the developers in order to execute the *Acceptance Tests*.

(3) Make acceptance tests fail

According to the traditional TDD approach, the tests are written before the actual code. This causes the new *Acceptance Tests* to fail when they are executed for the first time.

(4) Make acceptance tests pass

The development of the new feature starts following the traditional *Unit Test Driven Development* (UTDD) approach until all *Acceptance Tests* pass.

(5) Refactor

Once all Acceptance Tests pass, the code is refactored to improve the design and structure of the code. This helps to get rid of duplicate code as well as to improve comprehensibility.

(6) Customer accepting system

The customer reviews the system and reruns the *Acceptance Tests* in order to verify if it meets his expectations. If changes have to be made to the system, the *Acceptance Tests* are adjusted accordingly and the cycle starts from the beginning.

Refactoring has been mentioned in conjunction with EATDD and represents a very important technique in TDD in general. Chapter 3.4 gives more information about the definition of *Refactoring* and explains how *Refactorings* are related to *Acceptance Tests*.

3.4 Refactoring Of Acceptance Tests

Refactoring in general is a special software development technique used to change the internal structure of a software system without changing its external behaviour. *Fowler* defined refactoring 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 et al. 2000]".

Refactoring has the following purposes [Fowler et al. 2000]:

- **Improving the design of software**

  With proceeding development of a system, more code will be added and changed which, most notably, causes the code to lose its structure. Thus, the design of the system will decay if the code is not refactored.

- **Making software easier to understand**

  In bigger software projects where different people participate in the development of a software system, the same piece of code is written and modified by more than one person. In order to make changes to another programmer's code, it is important to fully understand the structure and meaning of the code before it is modified. That is why it is important to keep the code as simple as possible, so other project members

or future developers will be able to understand it. Refactoring can help to keep code readable and understandable if it is executed regularly after implementing new functionality.

- **Finding bugs**

  Refactoring requires having an understanding of what the code does; otherwise, code could not be restructured. Since the refactoring process helps to understand the code better, it can also help to find bugs.

As mentioned in chapter 3.3, refactoring is an integral part of the development process of TDD. It is typically executed after the simplest thing is done to make a newly added test pass. Since TDD requires having a safety net of tests that covers the entire functionality of the system, confidence can be achieved that no defects were introduced in the course of the refactoring [Astels 2003].

In conjunction with acceptance tests, refactoring is used to apply changes to the test specification in such a way that the test and the corresponding *Fixture* remain consistent and the acceptance test can still be executed. More precisely, “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 the test execution results in either success, [...] or fail [...] [Ordelt 2008]”.

Through acceptance test refactoring, the effort needed to maintain acceptance tests can be reduced as well as the error-proneness that goes along with manual modification of acceptance tests because the test specification and the *Fixture* are kept consistent automatically.

3.5 GreenPepper Acceptance Tests

GreenPepper is an “Agile Requirements Definition and Management (RDM) tool. In addition to a conventional RDM, it also allows for verifying that the system accurately satisfies the requirements [Pyxis Paper]”. More precisely, it is a tool that helps integrating executable acceptance tests – also known as executable specifications – into software development processes.

The tool is developed by Pyxis Technologies whose headquarters is located in Montreal, Canada. Founded in 2000, the company focuses on providing tool support, coaching and training services for software developers, who like to apply an agile software development approach [Pyxis].

GreenPepper is described by Pyxis as "[...] the tool in which you implement the concept of executable specification" [GP FAQ]. It consists of a set of tools that help to design and maintain executable acceptance tests during the development of a system or application. One part of *GreenPepper* is an engine (*GreenPepper Runner*) to execute acceptance tests under different environments such as .NET and Java.

The engine requires the acceptance tests to have a specific standardized format, so that the content of the acceptance tests can be interpreted and mapped to the *System Under Test* (SUT) accordingly. The following chapters will explain the layout and structure of acceptance tests which are used by *GreenPepper* [GP Home].

3.5.1 Notation And Layout

Since this thesis deals with refactoring of *GreenPepper* acceptance tests, it is important to fully know and understand the structure and meaning of them. Therefore, this chapter will explain how test documents containing *GreenPepper* acceptance tests look like and how these tests are related or mapped to the SUT.

Two different notation formats

GreenPepper acceptance tests are noted down within common HTML files. There are two different ways of expressing them:

- in HTML table format
- in HTML bullet list format or HTML number list format

In the first case, a *GreenPepper* acceptance test is identified by enclosing HTML table tags. All text that is enclosed between a starting and closing table tag is considered to be part of an acceptance test. In other words, a starting table tag (written as <table>) designates the

beginning of a new acceptance test whereas an ending table tag (written as </table>) designates the end of the acceptance test. Nested HTML tables do not represent multiple acceptance tests, but are simply treated as plain content of just one acceptance test. How this content is interpreted will be explained in the following chapters.

Similar to the table format, *GreenPepper* acceptance tests can also be noted down as bullet lists or number lists. Instead of being enclosed by table tags, the acceptance test is surrounded by opening and closing HTML list tags. This can either be the "unordered list" tag (written as <ul> or) or the "ordered list" tag (written as <ol> or). As for table tags, nested HTML list tags do not represent multiple acceptance tests, but will also be interpreted as content of just one acceptance test.

All text within the HTML file that is located outside the above described special HTML tags is ignored and therefore not treated as an acceptance test. This makes it possible to add comments or other useful information to the *GreenPepper* acceptance test files.

Equivalent notation

Compared to the table format, the bullet list or number list notation is an equivalent way of writing *GreenPepper* acceptance tests down. Having said that, it makes no difference whether an acceptance test is noted down in list format or table format and it will not change the meaning or interpretation of the test. That is why the following chapters will explain the structure of *GreenPepper* acceptance tests using the example of the table format.

Interpreter types

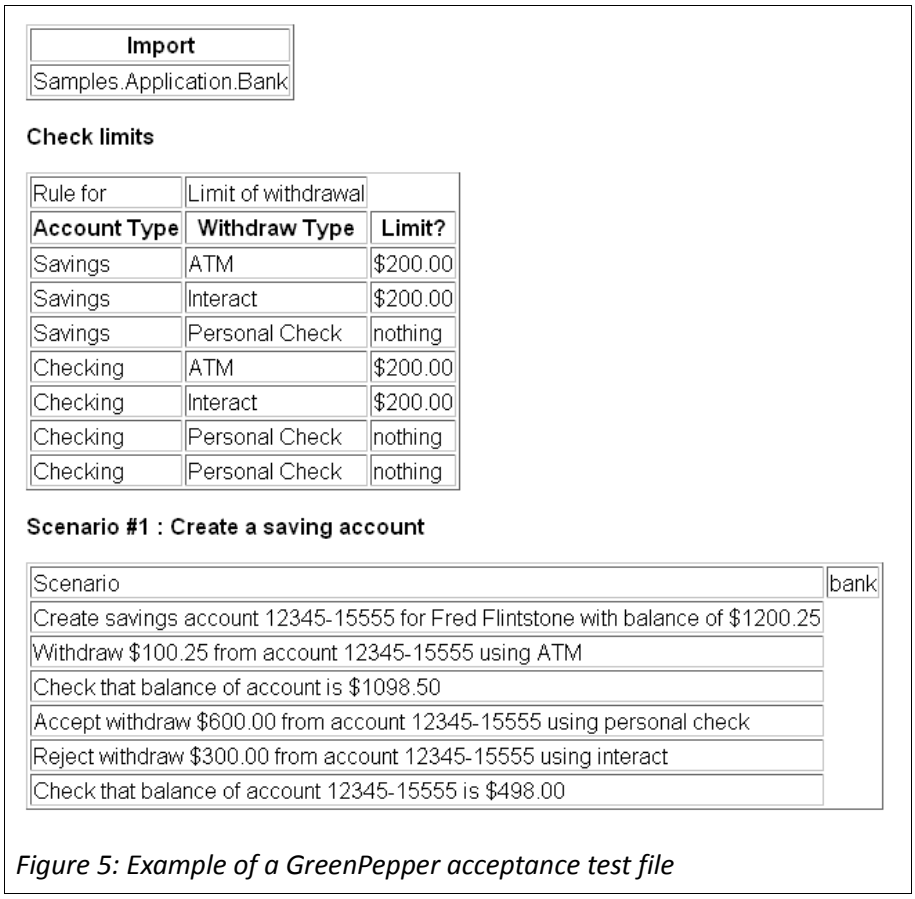
As mentioned before, a *GreenPepper* acceptance test is expressed as a common HTML table. The table must have a special format in order to represent a valid *GreenPepper* acceptance test.

The first cell in the first row of the table always specifies the so-called interpreter type. The interpreter defines how the remaining cells of the table are interpreted and what kind of acceptance test is represented by the table. The following list shows all available interpreters:

- *Import* interpreter

- *RuleFor* interpreter
- *Scenario* interpreter
- *Info* interpreter
- *Comment* interpreter
- *DoWith* interpreter
- *List* interpreters (*ListOf*, *SetOf*, *SubsetOf*, *SupersetOf*)
- *SetUp* interpreter [GP Doc]

Figure 5 demonstrates a typical *GreenPepper* acceptance test file. The file contains three tables in total, whereas each of them represents an acceptance test which is indicated by the interpreter name in the first cell of the first row. The descriptive text between these tables is not interpreted since it is not contained within a HTML table.



Import
Samples.Application.Bank

Check limits

Rule for	Limit of withdrawal	
Account Type	**Withdraw Type**	**Limit?**
Savings	ATM	$200.00
Savings	Interact	$200.00
Savings	Personal Check	nothing
Checking	ATM	$200.00
Checking	Interact	$200.00
Checking	Personal Check	nothing
Checking	Personal Check	nothing

Scenario #1 : Create a saving account

Scenario	bank
Create savings account 12345-15555 for Fred Flintstone with balance of $1200.25	
Withdraw $100.25 from account 12345-15555 using ATM	
Check that balance of account is $1098.50	
Accept withdraw $600.00 from account 12345-15555 using personal check	
Reject withdraw $300.00 from account 12345-15555 using interact	
Check that balance of account 12345-15555 is $498.00	

Figure 5: Example of a GreenPepper acceptance test file

The example comprises the *Import* interpreter, *RuleFor* interpreter and *Scenario* interpreter. These, as well as the *Comment* and *Info* interpreters, are explained in greater detail in the following chapters, whereas the other interpreter types are described only roughly. The reason is that this thesis focuses only on refactoring of *RuleFor* acceptance tests as well as *Scenario* acceptance tests, which makes it more important to fully understand them rather than the remaining interpreter types. The *Import* interpreter also plays a major role when it comes to refactoring of acceptance tests, which is why it is also explained in greater detail.

3.5.1.1 RuleFor Interpreter

Usage

The *RuleFor* interpreter is "[...] used to express concrete and measurable business rules" [GP Doc]. It allows for specifying a number of given values and expected values. When the *RuleFor* acceptance test is run by the *GreenPepper* engine, the engine calculates the results based on

the given values by calling the SUT and compares them to the expected values specified in the test.

A good example might be a Calculator, which calculates the quotient of two numbers. Given values would be the dividend and the divisor, the expected value would be the quotient. Figure 6 shows an example of how a *RuleFor* test would look like in order to test the correct behaviour of the calculator in operating a division.

rule for	Division	
dividend	**divisor**	**quotient?**
6.0	2.0	3.0
7	2	3.5
0	3	0

Figure 6: RuleFor test for a calculator

Structure

Figure 7 shows the general structure of a *RuleFor* test. As described earlier, the first cell of the first row specifies the interpreter type, which is “rule for” in case of a *RuleFor* test. It is followed by the name of the test that identifies the set of rules. The name can be chosen randomly by the business man writing the test, but should describe the objective of the test in a meaningful way. Furthermore, the name of the test is mapped to the Fixture, which is explained in chapter 3.5.2.

	rule for	*Identification of the set of rules*						
Header →	*Param 1*	*Param 2*	…	*Param N*	*Question 1* ?	*Question 2* ?	…	*Question M* ?
Example 1 →	given 1.1	given 1.2	…	given 1.N	expected 1.1	expected 1.2	…	expected 1.M
Example 2 →	given 2.1	given 2.2	…	given 2.N	expected 2.1	expected 2.2	…	expected 2.M
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
Example Y →	given Y.1	given Y.2	…	given Y.N	expected Y.1	expected Y.2	…	expected Y.M

Figure 7: Structure of a RuleFor test [GP Doc]

The second row is called the header row and serves to distinguish between given and expected values. If a header name ends with either “?” or “()”, it denotes an expected value. The given values serve as input values for the calculation that is taking part within the SUT, whereas the expected values serve as comparison values against those returned by the SUT [GP Doc].

All following rows contain concrete examples for the given and expected values. As for the calculator example shown in Figure 6, the third table row applies a value of “6.0” and “2.0” to the given values named “dividend” and “divisor”. The expected value “quotient” is assigned a value of “3.0” since this is the expected result of the division operation.

3.5.1.2 Scenario Interpreter

Usage

The *Scenario* interpreter “[...] is used to express interactions with the system under development that must be performed in a particular order [GP Doc]”. Therefore, it makes it possible to test the dynamic behaviour of the SUT for a sequence of actions. A big advantage of the *Scenario* interpreter is that the actions can be written down in natural language.

An example of a *Scenario* test for a simple bank application can be seen in figure 8. The notional system supports standard bank account functions such as the opening of a checking account and the deposit and withdrawal of money. The test describes possible interactions with the system in a particular order: A new checking account is opened before money is deposited and withdrawn.

Scenario	bank
open checking account 12345-67890 under the name of Spongebob Squarepants	
verify that balance of account 12345-67890 is $0.00	
deposit $100.00 in account 12345-67890	
verify that balance of account 12345-67890 is $100.00	
withdraw $50.00 from account 12345-67890	
verify that balance of account 12345-67890 is $50.00	
can't withdraw $75.00 from account 12345-67890	
verify that balance of account 12345-67890 is $50.00	
can withdraw $25.00 from account 12345-67890	

Figure 8: Scenario test for a bank application

Structure

In order to indicate a *Scenario* test, the first cell of the first row must be labeled with “Scenario”. Similar to the *RuleFor* interpreter type, the next cell contains the name of the test.

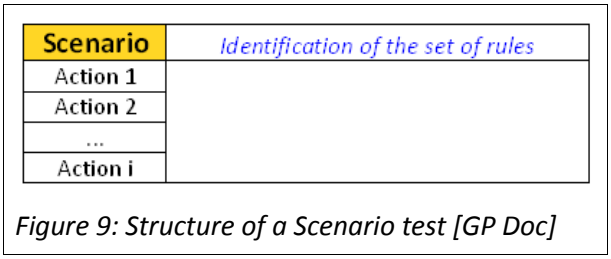


Figure 9: Structure of a Scenario test [GP Doc]

As can be seen in figure 9, the test layout continues with an arbitrary number of rows. Each row consists of only one cell, which holds the text of an action that should be performed on the SUT when the test is executed. There is no need for additional formatting of the actions. As mentioned before, the actions can be written down in natural language, which is illustrated by the example in figure 8.

3.5.1.3 Import Interpreter

Each acceptance test is mapped to a so called *Fixture*, which intermediates between the acceptance test and the SUT. Detailed information about *Fixtures* can be found in chapter 3.5.2.

The mapping to a *Fixture* requires having full qualified class names that consist not only of the class name itself but also of the namespace. This mechanism is used by the .NET framework to include two different classes with equal names into the same project.

As far as GreenPepper acceptance tests are concerned, the namespace is used to identify the exact location of a *Fixture* within a project. Since namespaces can have very long, not easy to read names, it is possible to define an *Import* interpreter within a *GreenPepper* acceptance test file. It works similar to the “using” statement of the C# programming language (see chapter 3.6.1.1). As can be seen in figure 10, it simply allows for noting down a list of different namespaces that are automatically used for the *Fixture* mapping process. As a result, these namespaces do not have to be noted down again anywhere in the same acceptance test file.

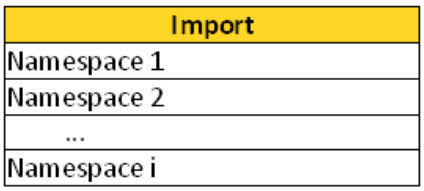


Figure 10: Structure of an Import interpreter [GP Doc]

More information about the *Import* interpreter can be found in chapter 3.5.2.

3.5.1.4 Info And Comment Interpreter

As explained at the beginning of chapter 3.5.1, common HTML tables represent *GreenPepper* acceptance tests. That means, the GreenPepper engine tries to interpret each HTML table within a test file as an acceptance test. This behaviour can cause problems when additional information (comments) needs to be provided in the form of HTML tables.

To solve this problem, both the *Info* and *Comment* interpreter can be used. The use of these interpreters prevents HTML tables from being interpreted as acceptance tests.

Info interpreter

As can be seen in figure 11, the *Info* interpreter consists of two parts. Each part is a HTML table with just one cell holding one of two possible *Info* interpreter keywords: “Begin Info” and “End Info”.

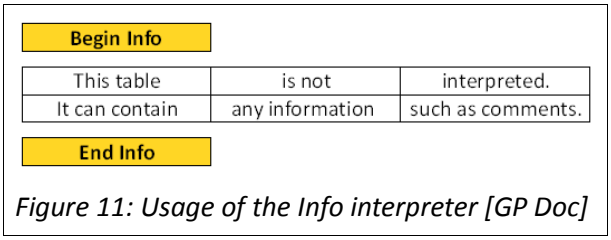


Figure 11: Usage of the Info interpreter [GP Doc]

“Begin Info” tells the *GreenPepper* engine to skip all HTML tables for execution until the “End Info” interpreter keyword is read. That means, all content (tables) between the “Begin Info” and “End Info” table will be ignored and not interpreted as *GreenPepper* acceptance tests. If the “End Info” table is omitted, all content of the entire file after the “Begin Info” table is

skipped [GP Doc].

Comment interpreter

Figure 12: Usage of the Comment interpreter [GP Doc]

The *Comment* interpreter is used to comment existing *GreenPepper* acceptance tests out, so that they are not interpreted anymore. Figure 12 shows how this is done: The old table representing an acceptance test (labeled as “My table” in the figure) needs to be wrapped by another table with four cells. In doing so, the third cell contains the whole acceptance test. The first cell is marked with the keyword “Comment” whereas the wrapping table ends with the keyword “to skip” in the fourth and last cell.

3.5.1.5 Other Interpreters

For the sake of completeness, this chapter will give an overview of all remaining interpreter types that have not been described yet. As mentioned before, these interpreter types are not essential for the work described in this thesis, but will help better understand *GreenPepper* acceptance tests. For detailed information about these interpreter types, the documentation homepage can be consulted (see [GP Doc]).

DoWith interpreter

The *DoWith* interpreter is very similar to the *Scenario* interpreter (see chapter 3.5.1.2). In fact, it can be regarded as its ancestor. It also allows for specifying a sequence of actions that is executed in a particular order on the SUT. A *DoWith* test is denoted with the “do with” keyword in the first cell of a table [GP Doc].

List interpreters

List interpreters are used to “express any kind of group, list or set of values [GP Doc]“. They are very helpful to check if a collection within the SUT corresponds to an expected collection of values. In other words, it is possible to compare two different collections or lists of values. The

difference between a collection and a list is that the order of elements does not matter in case of a collection. A list, on the contrary, respects the order of its elements.

There are four different *List* interpreter types:

- *ListOf* interpreter
- *SetOf* interpreter
- *SubsetOf* interpreter
- *SupersetOf* interpreter

The *ListOf* interpreter makes sure that the list provided in the test specification matches the content *and* the order of the list returned by the SUT, whereas the SetOf interpreter only matches the content. The *SubsetOf* interpreter verifies that the list provided in the test specification represents a subset of the list returned by the SUT, whereas the *SupersetOf* interpreter operates exactly in the opposite way.

As for all other interpreter types, the *List* interpreters are denoted with their respective keywords in the first cell of a table [GP Doc].

SetUp interpreter

The *SetUp* interpreter is used to "simplify the creation of a particular state for the system under development [GP Doc]". Starting with the "Set Up" keyword in the first cell of a table, a *SetUp* test provides data that is inserted in the SUT and which is needed to conduct subsequent tests.

Thinking of the bank example introduced in chapter 3.5.1.2, there must already exist two accounts in order to test a transfer action between these accounts. In this example, the *SetUp* interpreter could be used to initialize the state of two existing accounts.

3.5.2 Fixtures

Like chapter 3.5.1 explained, *GreenPepper* acceptance tests are noted down within common HTML tables that provide a clear and human readable way of specifying test scenarios for a system. The *Scenario* interpreter, in particular, allows for using natural language to describe an

action that is performed on the SUT. The SUT, on the other hand, is implemented in a programming language such as C#, which differs a lot from natural language and is used by the developers of the system.

Fixtures are used to intermediate between these two languages or the test specification and the SUT respectively (see figure 13). They are written by the developers in the same programming language as the system, whereas the developers are responsible for making the *Fixture* call the appropriate functions of the SUT as intended by the acceptance test. In other words, a *Fixture* represents the interpretation of an acceptance test and performs all actions on the SUT which are expressed by the test.

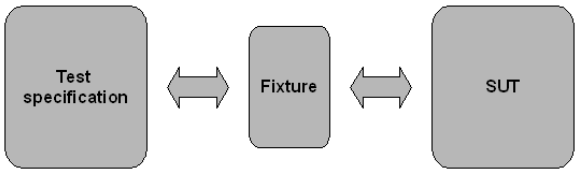


Figure 13: Fixture as mediator between test specification and SUT

In the following, the correlation between *GreenPepper* acceptance tests and *Fixtures* is explained on the basis of the C# programming language. The explanation focuses on two interpreter types:

- *RuleFor* interpreter (see chapter 3.5.2.1)
- *Scenario* interpreter (see chapter 3.5.2.2)

Fixture class

Each *GreenPepper* acceptance test or interpreter (*RuleFor* or *Scenario*) is mapped to one *Fixture*. Since C# is an object-oriented programming language, a *Fixture* is represented by a class. That means that each acceptance test is mapped to a C# class. The name of the corresponding class or *Fixture* is determined by the name of the test.

Camel-Casing

There are several naming conventions for C# class names. For example, the class name should

start with a capital letter and must not contain spaces. Since these conventions lead to less readable names (especially those which consist of more than one word), *GreenPepper* uses a so called *Camel-Casing* mechanism, which makes it possible to provide more comprehensible names within the test specification. Instead of using the exact test name for the mapping, this mechanism is applied to find the correct matching *Fixture* class.

In order to clearly distinguish between classes that belong to the system and classes that serve as *Fixture,* it is possible to add the suffix "Fixture" to the *Fixture* class names. *GreenPepper* will implicitly add the suffix to the test name if necessary. Table 2 gives some examples of possible mappings of test names and *Fixture* names.

Test name	Corresponding Fixture name
a very long name	AVeryLongName
a very long name	AVeryLongNameFixture
bank	Bank
bank	BankFixture
Bank	Bank

Table 2: Mapping of test name and Fixture name

Namespaces and Import interpreter

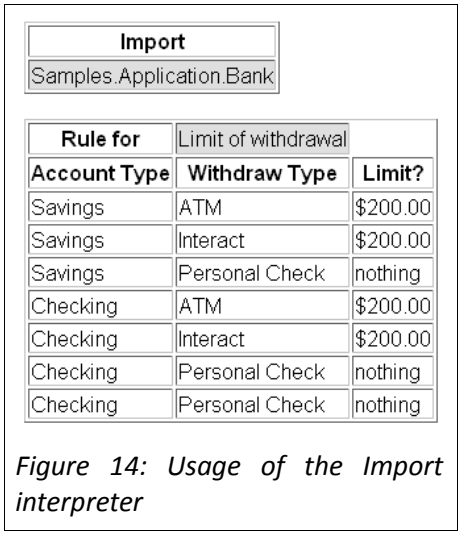
The name of a class is not sufficient for referencing the class clearly. C# uses a logical structure called namespaces to solve the problem of ambiguity that is caused by multiple classes which have the same name. Each class belongs to a namespace whereas classes with equal names must be within different namespaces.

In order to locate a *Fixture* class explicitly, the respective namespace must also be specified in the test specification. There are two possible ways to do so:

(1) Provide the corresponding namespace with the test name. The test name represents the full qualified name of the *Fixture* class.

(2) Use the *Import* interpreter to import namespaces and implicitly attach them to the *Fixture* class name. If that approach is taken, the test name does not have to contain

the namespace, which increases readability. In the example shown in figure 14, the *RuleFor* test with the name “Limit of Withdrawal” would be mapped to the Fixture class “LimitOfWithdrawal” within the namespace “Samples.Application.Bank”.



Import
Samples.Application.Bank

Rule for	Limit of withdrawal	
Account Type	**Withdraw Type**	**Limit?**
Savings	ATM	$200.00
Savings	Interact	$200.00
Savings	Personal Check	nothing
Checking	ATM	$200.00
Checking	Interact	$200.00
Checking	Personal Check	nothing
Checking	Personal Check	nothing

Figure 14: Usage of the Import interpreter

Fixture instantiation

Whenever a *GreenPepper* acceptance test is executed, the *GreenPepper* engine applies the above mentioned mapping mechanism to find the corresponding *Fixture*. Once it is found, the *Fixture* class is instantiated with the standard constructor by the *GreenPepper* engine using reflection. The created instance is used subsequently to perform the actions on the SUT as specified in the acceptance test.

Summary

This chapter explained how the *Fixture* class is determined that corresponds to a GreenPepper acceptance test and that is responsible for carrying out the actions specified in the test specification.

The following two chapters will focus on how the contents of the *RuleFor* and *Scenario* tests are mapped to the *Fixture* class and how the test specification interacts with the instantiated *Fixture* class.

3.5.2.1 RuleFor Interpreter

As described in chapter 3.5.1.1, a *RuleFor* test consists of columns for given values and expected values. The expected values represent the expected result of the calculation that is performed on the SUT using the given values as input parameters for the calculation. Figure 15 shows an example of a *RuleFor* test for the division operation of a calculator.

rule for	Division	
dividend	**divisor**	**quotient?**
6.0	2.0	3.0
7	2	3.5
0	3	0

Figure 15: Example for a RuleFor test

Mapping of given values

Each given value is mapped to a public field in the *Fixture* class (see figure 16). The name of the field corresponds to the header name of the given value in the test specification (*dividend* and *divisor*). As part of the name mapping, the *Camel-Casing* mechanism (see chapter 3.5.2) is applied to satisfy the naming conventions of C# fields.

The type of the field is not explicitly defined and can be chosen randomly by the developer based on the domain of the given value. The *GreenPepper* engine will automatically convert between different types. In case of the example above, a type of *double* would be reasonable, since the given values represent real numbers.

Mapping of expected values

Expected values imply a calculation being performed, so their value can be compared to the result calculated by the SUT. This is why each expected value is mapped to a public method (see figure 16). The method does not have any parameters. In fact, the input parameters or given values that are needed to perform the calculation are provided by the public fields and will be referenced within the method.

The name of the method is related to the header name of the expected value in the test specification (*quotient*). As for given values, the *Camel-Casing* mechanism is also used to apply the mapping.

The return type of the method depends on the domain of the expected value and is not defined explicitly. In the previous example, a type of *double* would be reasonable, since a division operation of two real numbers results in a real number.

Interactions between Fixture and test specification

Figure 16 shows the *Fixture* class that corresponds to the above mentioned *RuleFor* test example.

```
1  public class DivisionFixture {
2
3      public double dividend;
4      public double divisor;
5
6      public double Quotient() {
7          //call to SUT to calculate quotient
8          Calculator calc = new Calculator();
9          double result = calc.Divide(dividend,divisor);
10
11         return result;
12     }
13 }
```

Figure 16: Corresponding Fixture

The execution of the test is carried out by the *GreenPepper* engine in the following steps:

(1) The *RuleFor* interpreter is identified

(2) The *Fixture* class "DivisionFixture" is identified through the test name and instantiated with the default constructor.

(3) The columns for given values and expected values are identified.

(4) The value 6.0 is assigned to the field variable *dividend*.

(5) The value 2.0 is assigned to the field variable *divisor*.

(6) The method *Quotient()* is called to get the value calculated by the SUT.

(7) The value 3.0 is compared against the value returned by the *Fixture*. The cell is coloured appropriately (see chapter 3.5.3).

(8) Steps 4-7 are repeated accordingly for the remaining two rows.

3.5.2.2 Scenario Interpreter

A *Scenario* test consists of one or more actions that are performed on the SUT in a particular order (see chapter 3.5.1.2). The actions are noted down in natural language. Figure 17 shows an exemplary *Scenario* test, which specifies actions for a notional bank application.

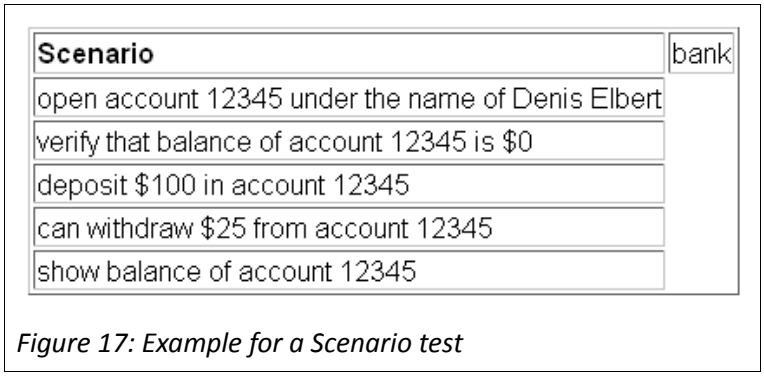


Figure 17: Example for a Scenario test

Each action is mapped to a public method within the *Fixture* class (see figure 18), whereas the mapping is not carried out based on the name of the method, as it is done in case of an expected value of a *RuleFor* test.

Mapping through regular expressions

Instead, each method is annotated with a special C# attribute that comes with the *GreenPepper* framework (see chapter 3.6.1.2 to learn more about C# attributes). The attributes have one parameter expecting a regular expression. The regular expression (see chapter 3.6.1.3) is used to relate the method to the action specified in the test specification. In order to find the corresponding method to an action, the *GreenPepper* engine searches for a regular expression within the *Fixture* that matches the text of an action. If a regular expression matches, the method that was annotated with this regular expression will be mapped to the action and executed by the *GreenPepper* engine.

```
using GreenPepper.Interpreters.Scenario;

public class BankFixture {

    private Bank bank;

    public BankFixture {
        bank = new Bank();  //initialize bank with SUT
    }

    [Given("open account (\\d{5}) under the name of ([\\w|\\s]*)")]
    public void OpenAccount(string accountNr, string name) {
        //call to SUT
        bank.OpenAccount(accountNr,name);
    }

    [Then("verify that balance of account (\\d{5}) is \\$(\\d+)")]
    public void BalanceOfAccount(string accountNr, Expectation expectedBalance) {
        //call to SUT
        expectedBalance.Actual = bank.GetAccountBalance(accountNr);
    }

    [When("deposit \\$(\\d+) in account (\\d{5})")]
    public void Deposit(double amount, string accountNr) {
        //call to SUT
        bank.Deposit(amount,accountNr);
    }

    [Check("can withdraw \\$(\\d+) from account (\\d{5})")]
    public bool CanWithdraw(double amount, string accountNr) {
        //call to SUT
        return bank.AmountIsAvailable(amount, accountNr);
    }

    [Display("show balance of account (\\d{5})")]
    public double ShowBalance(string accountNr) {
        //call to SUT
        return bank.GetAccountBalance(accountNr);
    }
}
```

Figure 18: Corresponding Fixture to the Scenario test

Method parameter mapping

Parameters are identified by regular expression grouping constructs, which are marked with left and right parentheses within the regular expression (see chapter 3.6.1.3). The captured sub-expression of each group is mapped sequentially to the parameters of the respective method. That means, the method must have as many parameters as the regular expression has group constructs, otherwise the mapping will fail.

The type of the method parameters can be chosen randomly by the developer based on the domain they belong to. In the antecedent example, it is reasonable to use a type of *double* for the parameter holding the amount of money.

Five attributes

There are several attributes (a total of five) used to annotate a method within the *Fixture*. They express different kinds of actions that can be performed on the SUT and have an influence on the signature of the corresponding method. Table 3 gives an overview of these attributes [GP Doc] and their influence on the method signature (method return type and parameters).

Attribute	Usage	Method return type	Additional method parameters
Given	To put the system in a known state before subsequent actions are performed.	void	none
Then	To verify the result of interactions with the system by comparing it to an expected value.	void	Expectation object
When	To bring the system to another state.	void	none
Check	To verify the result of an action.	boolean	none
Display	To show the result of an action. Only for informational purpose.	any type	none

Table 3: Different attribute types for a Scenario action

Expectation object

The *Then* attribute sets itself apart by requiring an *Expectation* object as a method parameter. The *Expectation* object is provided by the *GreenPepper* framework and is used to store the expected value as well as the actual result returned by the SUT. The *GreenPepper* engine uses this object to compare these values against each other. Whenever the *Then* attribute is used, the last sub-expression of the regular expression is mapped to this *Expectation* method parameter.

Standard interaction process

Disregarding the attribute, the execution of a *Scenario* test is carried out in the following basic steps by the *GreenPepper* engine:

(1) The *Scenario* interpreter is identified

(2) The corresponding *Fixture* class is identified through the test name and instantiated with the default constructor.

(3) Steps 4-5 are repeated for each action specified in the test.

(4) The *GreenPepper* attributes are identified and the regular expression is determined that matches the text of the actual action.

(5) The group constructs of the matching regular expression are retrieved and assigned sequentially to the parameters of the corresponding method. The method is called with these parameters.

3.5.3 Test Results

As described in the previous chapters, the *GreenPepper* engine combines each acceptance test with a *Fixture* that intermediates between the test specification and the SUT. The *Fixture* is written by the developers and carries out the actions on the SUT that are specified by the acceptance test.

After a test is run, the customer or developer needs to be notified whether the test was successful, i.e. the system met the requirements specified by the acceptance test, or not. The *GreenPepper* engine therefore compares the results returned by the *Fixture* with the expected values provided in the test specification. Then, it creates a copy of the original test specification document and marks the appropriate table cells of the respective part of the test with different colours. Four different colours are consistently used to indicate the test result. Based on the interpreter type that is used for an acceptance test, the interpretation of the colours may differ. Table 4 explains the meaning of each colour in case of a *Scenario* test and a *RuleFor* test.

Colour	Interpretation for *Scenario* test	Interpretation for *RuleFor* test
Green	The scenario action has been executed successfully.	The test has been executed successfully and the result returned by the SUT is in accordance with the expected value.
Red	The scenario action could not have been executed.	The test has been executed successfully, but the result returned by the SUT differs from the expected value.
Yellow	An exception occurred while running the test.	An exception occurred while running the test.
Gray	The scenario action has been executed successfully and the result returned by the SUT is displayed (the *Display* attribute must have been used, see chapter 3.5.2.2).	The test has been executed successfully and the result returned by the SUT is displayed (only when no expected value is specified in the test).

Table 4: Test result colouring for different interpreter types [GP Doc]

Figure 19 shows an exemplary test result as it is generated by the *GreenPepper* engine after the execution of a *GreenPepper* acceptance test.

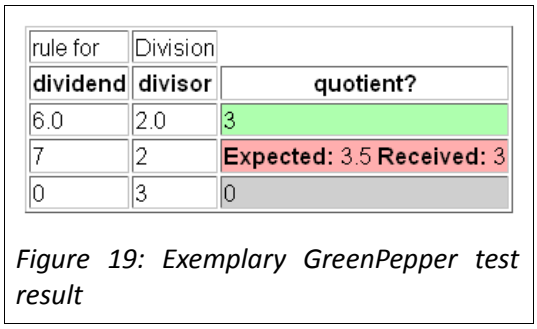


Figure 19: Exemplary GreenPepper test result

3.6 .NET Framework

.NET is a software framework developed by Microsoft. It consists of a comprehensive class library and a runtime environment and is used to develop, compile and execute applications on the Microsoft Windows platform. Some of the basic characteristics of the Microsoft .NET framework are listed and described below [Kuehnel 2008]:

- **Common Language Runtime**

  The *Common Language Runtime* (CLR) represents a virtual machine (similar to the Java virtual machine) for the .NET framework. All .NET applications are compiled to a byte

code called *Common Intermediate Language* (CIL), which is converted to native machine code during execution time by the CLR's *Just-In-Time* compiler (also referred as JITter).

- **Language independence**

  Through the definition of the *Common Language Specification* (CLS), .NET applications can be written in any .NET compliant language (such as C#, J#, C++, VB.NET). The CLS defines policies, which have to be observed by each language to ensure interoperability between these different languages. This is also supported by the *Common Type System* (CTS), which specifies all data types recognized by the CLR. As a result, all .NET compliant languages produce compatible CIL-code, which is independent from the programming language used.

- **Object-oriented**

  .NET is fully object-oriented and offers a consistent, logical infrastructure for developing applications. It also encapsulates functions of the Win32-API in classes to provide access to the Windows operating system.

- **Memory management**

  Through the introduction of the *Garbage Collector* (GC), not referenced memory is freed automatically in the background, so developers do not have to take care about this problem.

3.6.1 C#

C# is a programming language that was specifically designed for the .NET *Common Language Runtime* (CLR). Although it is possible to write .NET applications in other .NET compliant languages (such as C++ or VB.NET), the use of C# is more appropriate in most cases because it perfectly integrates into the .NET environment.

Furthermore, C# shares the same roots as C++ and Java and there are several syntactical elements which conform to each other. Thus, developers who are familiar with C++ or Java can still make use of their potential when using C# [Gunnerson 2000].

This chapter will describe some of the features of C#, which are relevant for this thesis.

3.6.1.1 Namespaces

A namespace is a logical, organizational structure which is used to assign a class to an appropriate subject area. This helps locating a class with specific functionality within a class library.

In addition to that, namespaces are necessary to solve ambiguities between class identifiers, which are represented by unique class names. The name is used to instantiate a new object and to access its functionality. Through the concept of namespaces, the class name has to be unique only within the same namespace. In other words, by identifying classes by their assigned namespace *and* their class name (also called *full qualified identifier*), two equally named classes can be assigned to different namespaces and therefore clearly referenced [Kuehnel 2008].

Instead of always providing the full qualified identifier when referencing a class in the source code, C# allows for the import of namespaces at the beginning of the source code file with the *using* keyword. This will cause the compiler to search for the classes within these namespaces.

3.6.1.2 Attributes

Attributes are a special feature of the .NET framework and can be compared with annotations in Java. They are used to provide additional information to code elements (e.g. a class, field or method) during runtime [Kuehnel 2008].

Figure 20 shows an example, in which the *Obsolete* attribute was used to mark a method as obsolete, which expresses that this method is still available only for compatibility reasons but should not be used in future time.

```
1  public class Math {
2
3      [Obsolete("Use method Sin2 instead.",true)]
4      public double Sin1(double x) {
5          // calc sin of x
6      }
7
8      public double Sin2(double x) {
9          //calc sin of x
10     }
11 }
```

Figure 20: Example for a .NET attribute

As can be seen in this example, an attribute is noted down within squared brackets right before the respective code element and may contain one or more parameters separated by commas. In this case, the first parameter defines an error message that shall be displayed when the first method is used whereas the second parameter tells the compiler to create an error rather than a warning.

Attributes are used in connection with *Scenario* acceptance tests (see chapter 3.5.2.2).

3.6.1.3 Regular Expressions

Regular expressions are used to process text. More precisely, they allow to "[...] quickly parse large amounts of text to find specific character patterns; to extract, edit, replace, or delete text substrings [...] [MSDN 2010a]".

They are processed by a regular expression engine which usually requires at least two inputs [MSDN 2010a]:

- A regular expression pattern that is to identify in the text. The pattern is written in a formal language that comes with a special syntax. The syntax is interpreted by the engine in order to match the pattern to the appropriate text substring.
- A text to parse for the regular expression pattern. The engine searches the text for the specified regular expression pattern and returns all matches of text substrings. These text substrings are also referred to as *captures*.

The .NET class library offers classes within the *System.Text.RegularExpressions* namespace to

process regular expressions. Due to the complexity of the regular expression implementation, this chapter describes only the concept of *grouping constructs*, which plays a major role when it comes to *GreenPepper Scenario* tests (see chapter 3.5.2.2). Detailed information about .NET regular expressions can be found in the *Microsoft Developer Network* (MSDN) at [MSDN 2010a].

Grouping constructs

Grouping constructs "[...] delineate sub-expressions of a regular expression and capture the substrings of an input string [MSDN 2010a]". Each group is enclosed by a left and right parenthesis and can thereby be distinguished from the residual part of the regular expression pattern. The regular expression pattern as a whole is also treated as a group. This special group captures the text that is matched to the whole regular expression pattern.

When the regular expression is processed, the regular expression engine numbers each group automatically based on the order of the opening parenthesis, starting from one. Group number zero is the special group representing the whole regular expression pattern. The group numbers are used to access the captures of the matched substrings of each group.

Regular expressions and grouping constructs play a major role when it comes to *Scenario* acceptance tests (see chapter 3.5.2.2).

3.6.2 Visual Studio

Visual Studio is an *Integrated Development Environment* (IDE) from Microsoft. It supports several high level programming languages such as C#, C++ and VB.NET and allows for the development of different kinds of applications for the Windows platform. The following application can be built with *Visual Studio* [Avery 2005]: Console applications, Windows forms applications, Windows services, dynamic web applications, web services, Windows mobile applications and Win32 applications.

It is also possible to extend *Visual Studio* by new functionality. Visual Studio offers three different methods to add new functionality to the IDE:

- **Macros**

Macros are used to automate a sequence of tasks within the IDE, so that the developer does not have to execute them manually. They are especially useful when the same sequence of tasks has to be executed frequently and in short time iterations. Since macros only afford automating recurring tasks, they represent the least powerful method to extend *Visual Studio*.

- **Add-ins**

  Add-Ins use high level APIs from *Visual Studio* such as the Development Tools Extensibility (DTE) object-model in order to manipulate default IDE windows (e.g. tasks list, output list, error list) or code displayed with the Visual Studio editor. Due to the use of high level APIs, *Add-ins* have some limitations and are less powerful than VSPackages, [Nayyeri 2009b], [Nayyeri 2009d].

- **VSPackages**

  VSPackages use low level APIs from *Visual Studio* and integrate into *Visual Studio* like a built-in part of the IDE. Thus, they are not limited in scope like *Add-ins* and allow for manipulating all kinds of user interface elements of the IDE such as menu bars, context menus, solution explorer, class view and more [Nayyeri 2009c], [Nayyeri 2009d].

All three methods are provided by the *Visual Studio* SDK, which is a framework used for extending the Visual Studio. The following chapters describe selective tools provided by the Visual Studio SDK, which are used to extend the IDE in the course of this thesis.

The most recent version of Visual Studio is the release candidate of Visual Studio 2010 and was published on February, 6th 2010.

3.6.2.1 Development Tools Extensibility (DTE)

Along with the *Visual Studio* SDK, which was mentioned above, *Visual Studio* features a programming model for extending and automating the IDE known as the *automation model*. The highest level object in the automation model hierarchy is the *DTE* object. It represents the *Visual Studio* IDE and allows for programmatically controlling and extending the IDE [MSDN 2010d], [MSDN 2010e].

The DTE object was used to interact with the *Visual Studio* editor (see chapter 5.3).

3.6.2.2 VSCT Files

A *Visual Studio Command Table* (VSCT) file is a special XML-file that describes the set of commands a *VSPackage* (see chapter 3.6.2) contains. The file is compiled by the VSCT compiler into a binary file whenever the *VSPackage* is loaded into *Visual Studio* for the first time [MSDN 2010f].

VSCT files distinguish between four different command types:

- **Buttons and Combos**

  Buttons and combos are the commands that a user can see and interact with. They are assigned to a group.

- **Groups**

  Multiple buttons and combos can be grouped together. A group always belongs to a menu.

- **Menus**

  Menus contain one or more groups.

The VSCT file is divided into four different sections:

- **Commands section**

  In this section the different menus, groups, buttons and combos are specified along with their properties (visibility, icons, ...).

- **CommandPlacements section**

  This section specifies the relation between the commands defined in the commands sections, that is, how to arrange and place the commands.

- **Bitmaps section**

  This section defines the bitmaps that are used for the commands.

- **Symbols section**

Each command is identified through an unique ID. This sections specifies all Ids that are used within the VSCT file.

As part of this work, a VSCT-file was used to incorporate the required refactoring commands within *Visual Studio* (see chapter 5.2).

3.6.3 Windows Presentation Foundation (WPF)

Windows Presentation Foundation (WPF) is a framework for developing graphical user interfaces for applications running on Windows platforms. It is part of Microsoft's .NET framework since it was introduced with version 3.0.

WPF represents an alternative to the “traditional” *Windows Forms* application programming interface (API) and is characterized by the following features [Kuehnel 2008]:

- The specification of the design and layout of the user interface can be completely separated from the code implementing the logic. This is established by describing the layout with XAML (eXtensible Application Markup Language), a language that is derived from the Extensible Markup Language (XML).
- Graphical user interfaces (GUI) that have been implemented using WPF, can be displayed within a common application window as well as in a web browser.
- WPF supports 2D- and 3D-graphics as well as animations, videos, images and audio files.
- WPF supports data binding.

WPF was used to implement the graphical user interface for the refactorings (see chapter 5.4).

4 Concept

This chapter describes the general idea of how the thesis goals defined in chapter 1.2 were accomplished. In summary, the overall objective was to implement refactoring functionality for *GreenPepper* acceptance tests within Visual Studio 2010.

Before the implementation could take part, several major decisions had to be made in order to plan each step that is necessary to achieve all the objectives. The preliminary considerations were focused on the following questions:

- How to extend Visual Studio 2010 by refactoring commands?
- How to refactor *RuleFor* and *Scenario* acceptance tests, i.e. which changes have to be applied to the acceptance test file and the *Fixture*?
- How to apply changes to an acceptance test?
- How to apply changes to a *Fixture*?
- Which user inputs are needed for each kind of acceptance test refactoring and how to obtain them?

Answers to all these basic questions are given throughout this chapter.

4.1 Extending Visual Studio 2010

Context menu

The thesis goals (see chapter 1.2) define a total number of seven diverse refactorings which were to perform on *GreenPepper* acceptance tests. These refactorings had to be incorporated into the Visual Studio 2010 IDE.

The Visual Studio SDK (see chapter 3.6.2) allows for the modification of several user controls of Visual Studio 2010 such as context menus, menu bars, command bars, tool bars, modification of windows such as the solution explorer view or class view or creating and integrating of own views. However, it was necessary to identify the most appropriate way of integrating the new

refactoring functionality.

Since Visual Studio 2010 comes with several source code refactoring functions (e.g. rename, extract method, encapsulate field and more) that are available through the context menu of its inbuilt editor, it was most appropriate to provide all acceptance test related refactoring functions within the same context menu (but in another submenu). One advantage of this approach is that a user who is already familiar with source code refactorings in Visual Studio 2010 will be able to use acceptance test refactorings in the same manner, thus, the workflow is identical.

GreenPepe2010

As mentioned before, context menus in Visual Studio 2010 can be modified by the use of the Visual Studio SDK. The SDK offers three different ways of extending Visual Studio (Macros, Add-Ins, *VSPackages*), whereas *VSPackages* (also referred to as *Integration Packages*) represent the most powerful way since they also allow for accessing low level APIs of the Visual Studio IDE [MSDN 2010b][Nayyeri 2009c].

There already existed an extension for Visual Studio 2010 called *GreenPepe2010*, which was developed in advance to this work. It is closely related to the topic of this thesis because it allows for the execution of *GreenPepper* acceptance tests and to display and manage test results within Visual Studio 2010. More details about *GreenPepe2010* can be found in chapter 2.2.

Since *GreenPepe2010* is implemented as a *VSPackage* and is also related to *GreenPepper* acceptance tests, this project was extended to incorporate the new acceptance test refactoring functionality into Visual Studio 2010 rather than implementing a new *VSPackage* from scratch. The given infrastructure of the *GreenPepe2010* extension was utilized to modify the context menu of the Visual Studio 2010 editor for the new refactoring commands, which is explained in chapter 5.2.

4.2 Refactoring Workflow

As mentioned above, the refactoring commands were made available to the user in the context

menu of the inbuilt editor of Visual Studio 2010. Since *GreenPepper* acceptance tests are noted down within common HTML files, they can be opened in either the code or the design view of the editor. The code view simply displays the HTML code whereas the HTML code is interpreted in the design view and visualized just like in a web browser. The context menu appears whenever the user performs a right-click on the editor and provides context-sensitive features. That means, the context menu contains different menu items depending on what element was right-clicked on. The current location of the cursor could be determined with the Visual Studio SDK, which is explained in greater detail in chapter 5.3.

With respect to the context-sensitive nature of the editor's context menu, a refactoring action is typically run in the following sequence of steps:

(1) The user opens the test specification in the Visual Studio 2010 design or code view.

(2) The user right-clicks on the acceptance test element he would like to refactor (e.g. the test name). In the opening context menu, the user selects the preferred refactoring action.

(3) Depending on the selected refactoring action, an accordant graphical user interface pops up where the user can provide inputs that are required to perform the refactoring (e.g. the new test name in case of a “Rename test” refactoring).

(4) A preview over all the changes in the test specification and the corresponding *Fixture* class (if available) is displayed.

(5) The refactoring is executed and all changes are applied to the acceptance test and the *Fixture*.

In order to find out which *GreenPepper* acceptance test element was selected and to populate the context menu with the appropriate refactoring commands, the current position of the cursor must be determined when the user right-clicks on the editor. How this was done is explained in chapter 5.3.

4.3 Parser For GreenPepper Acceptance Tests

The refactoring of *GreenPepper* acceptance tests required reading and modifying the test specifications. As mentioned in chapter 3.5, the acceptance tests are specified as HTML tables in common HTML files. This necessitated the implementation of a parser, which creates an object-model to abstract from the internal HTML structure and allows for easily accessing and modifying the test information stored within the HTML file.

As far as refactoring of acceptance tests is concerned, there were some requirements for the parser or rather the object-model created by the parser:

- The parser must support reading, modifying and adding of test data.
- The parser should not reformat the HTML code when modifying the test or adding new test data.
- The parser must support retrieving particular test elements based on their location within the file. This requires the parser to store position data along with the object-model.

In the course of this thesis, a parser for *GreenPepper* acceptance tests was developed that meets the requirements above. Its implementation is explained in chapter 5.1.

4.4 C# parser

As part of acceptance test refactoring, not only the test specification must be manipulated but also the *Fixture* code in order to keep the test specification and *Fixture* consistent. The Fixture is written in the C# programming language. Since C# - or programming languages in general - are structured following complex grammar rules, a parser is needed to wrap the code into an (code) object-model that allows for accessing and modifying the code in a comfortable and easy way.

Due to the complexity of such a parser it could impossibly be implemented by on one's own in the course of this work. As a result, a parser for C# had to be found which was implemented by a third party. With respect to the actions that had to be performed on the *Fixture* class for each

refactoring (see chapter 4.5), the following requirements had been identified for the parser:

- The parser must be capable of identifying all C# language constructs.
- The parser must be compliant with the official C# language specification named ECMA – 334.
- Code elements such as method declarations and field declarations must be explorable using the code object-model. That means, the parser must allow for discovering code declarations within a class.
- The parser library must be open source and should not require a license.

Table 5 gives an overview of all C# parsers that were found and examined during the research and juxtaposes their features. As can be seen in the table, only one C# parser met all of the specified requirements above: The “NRefactory” parser is part of the open source IDE *SharpDevelop,* which is developed as an alternative to Microsoft's *Visual Studio*. In order to make use of its functionality, the parser was extracted from the *SharpDevelop* project, compiled to a separate .NET class library and imported to the refactoring project part of this work.

Parser	Open source?	exhaustive object-model?	ECMA – 334 compliant?	Code discovery and editing capabilities?	Remarks
Visual Studio Code Model	(yes)	no	yes	yes	Does not parse content of method bodies.
.NET CodeDOM	yes	(yes)	yes	no	Can only be used to generate code, not for parsing existing code .
CS CODEDOM Parser	yes	no	unknown	yes	Does not parse content of method bodies.
Metapec C# Parser	no	yes	yes	yes	Parser library does not respect C# naming conventions.
SharpDevelop: NRefactory Parser	yes	yes	yes	yes	object-model uses visitor pattern.

Table 5: Comparison of different third-party C# parsers

4.5 Refactorings

This chapter explains in detail what actions had to be performed for each kind of *GreenPepper* acceptance test refactoring. Each refactoring is described uniformly in four steps:

(1) Motivation

The motivation describes in which situations the particular refactoring can be applied.

(2) Required inputs

Each refactoring requires specific user input. This part describes all user inputs that are required for a particular refactoring to be executed.

(3) Input validation

The data entered by the user must be verified to ensure that the refactoring can be executed without any errors. This section describes what kind of validations have to be performed on the input data and identifies possible error cases.

(4) Workflow

Since a refactoring causes changes to the test specification and the *Fixture*, this part explains systematically each step that is performed during the refactoring.

(5) Example(s)

At least one example is given to demonstrate the effects of a particular refactoring.

Remarks

As mentioned earlier in chapter 3.5.2, the *GreenPepper* engine applies a so-called *Camel-Casing* mechanism in order to transform a name within the test specification into a name used in the *Fixture* that follows the nomenclature of C# identifiers. Whenever the equality of names is considered throughout this chapter, the application of the *Camel-Casing* mechanism is assumed. For example, the name “bank account” is considered to be equal to “BankAccount” because of the *Camel-Casing* mechanism.

4.5.1 Rename Test

Motivation

Each *GreenPepper* acceptance test is assigned a name to distinguish it from others. The name is also used to find the corresponding *Fixture*, which usually has the same name as the acceptance test.

Whenever a test needs to be renamed, the corresponding *Fixture* must be renamed as well. Otherwise the test cannot be associated with a *Fixture* anymore and subsequent executions of the test will fail. The “Rename Test” refactoring can be used to change the name of an acceptance test and ensure that the associated *Fixture* is also renamed appropriately.

Required inputs

The “Rename Test” refactoring requires only the new name to be entered.

Input validation

Since the test name is mapped to the *Fixture* class name, the name must be a valid C# class identifier. Furthermore, the refactoring cannot be executed if a class with that new name

already exists. This has to be verified before the refactoring is executed.

Workflow

The following sequence of steps is executed when carrying out the “Rename Test” refactoring:

(1) Rename acceptance test to the new name.

(2) Rename *Fixture* class to the new name.

(3) If present, change the name of the constructor(s) of the *Fixture* to the new name.

(4) Change the file name of the *Fixture* to the new name.

Example

Figure 21 shows an example for a *RuleFor* test where both the test specification and the *Fixture* can be seen. The test as well as the corresponding *Fixture* class is named “Division”.

rule for	Division	
dividend	divisor	quotient?
6.0	2.0	3.0
7	2	3.5

```
1  public class Division {
2
3      public double dividend;
4      public double divisor;
5
6      public double Quotient() {
7          ...
8      }
9  }
```

Figure 21: RuleFor test example before "Rename Test" refactoring

In this example, the “Rename Test” refactoring is performed to change the test name to Calculator Division”. The result of the refactoring is shown in figure 22.

rule for	Calculator Division	
dividend	divisor	quotient?
6.0	2.0	3.0
7	2	3.5

```
1  public class CalculatorDivision {
2
3      public double dividend;
4      public double divisor;
5
6      public double Quotient() {
7          ...
8      }
9  }
```

Figure 22: RuleFor test example after "Rename Test" refactoring

As can be seen in the figure above, the test has been renamed to “Calculator Division” and the *Fixture* name has been updated appropriately, too.

4.5.2 Refactorings For RuleFor Tests

4.5.2.1 Add Given / Expected Value Column

Motivation

As explained in chapter 3.5.1.1, a *RuleFor* test is intended to check if computations are performed correctly by the SUT. Based on various input parameters, the result returned by the SUT is compared against an expected value.

Changing requirements during development may make it necessary to adjust input parameters or to introduce new calculations; thus, the test specification must be adjusted appropriately (in TDD the test specification is changed before any modifications to the code) by adding new columns for input parameters or expected values.

The “Add column” refactoring can be used to include new given or expected value columns in a *RuleFor* test.

Required inputs

In order to perform an “Add column” refactoring, the user has to provide three inputs:

(1) Column type

The column type decides whether to add a given value column or an expected value column to the test specification.

(2) Insertion position

The order of given and expected value columns in a *RuleFor* test is significant. All given parameters which serve as an input for the calculation that is triggered by an expected value must be specified before the expected value columns. Therefore, the user must specify where to insert the new given or expected value column.

(3) Column name

The header name of each given or expected value column indicates its interpretation and is also used to map the values to the *Fixture* (see chapter 3.5.2.1).

Input validation

The insertion position must be within a valid range. For example, if a *RuleFor* test contains three columns, the position must be in the range from “0” (first position) to “3” (last position).

As mentioned earlier in chapter 3.5.2.1, given values are mapped to a class field whereas expected values are mapped to a method within the corresponding *Fixture*. This is why the column name must represent a valid C# identifier. Furthermore, it must be verified that the *Fixture* does not contain a field (in case of a given value column to be added) or a method (in case of an expected value column to be added) with the same name as the new column.

Workflow

The “Add column” refactoring is carried out in the following sequence of steps:

(1) Add a new column at the specified position in the test specification. Name the header of the new column accordant to the specified column name.

(2) If the new column is a given value column, add a public field with a return type of “string” to the *Fixture* class. Add a TODO-comment connected to the new field.

(3) If the new column is an expected value column, add a public method with no parameters and the return type “object” to the *Fixture* class. Add a TODO-comment

and a “NotImplemented”-exception *throw* clause to the body of the method.

Example: Add expected value column

The following example shows a *RuleFor* acceptance test before and after an “Add expected column” refactoring has been performed.

Initially, the test contains only one expected value column to test the calculation of the quotient of two numbers (see figure 23).

rule for	Calculator	
a	b	quotient?
6.0	3.0	2.0

```
1  public class Calculator {
2
3      public double a;
4      public double b;
5
6      public double Quotient() {
7          ...
8      }
9  }
```

Figure 23: RuleFor test before "Add expected column" refactoring

In order to test the calculation of the product of two numbers as well, the “Add expected column” refactoring is applied. The new expected value column is named “product”.

As can be seen in figure 24, the new expected value column was created in the test specification and a new method “Product” was added to the *Fixture* class during the refactoring process.

rule for	Calculator		
a	b	quotient?	product?
6.0	3.0	2.0	

```
1  public class Calculator {
2
3      public double a;
4      public double b;
5
6      public double Quotient() {
7          ...
8      }
9
10     public object Product() {
11         // TODO: Implement test method
12         throw new System.NotImplementedException(
13             "This method needs to be implemented");
14     }
15 }
```

Figure 24: RuleFor test after "Add expected column" refactoring

The TODO-comment within the newly generated method serves as a reminder for the developers that this method has not been implemented yet. When executing the test, the exception thrown in the generated method will cause the test to fail, so that the customer is notified about the missing implementation, too.

4.5.2.2 Remove Given / Expected Value Column

Motivation

In the same way as calculation parameters have to be added because of changed requirements, parameters can also become obsolete during the development and need to be removed from the test specification.

The “Remove column” refactoring is used to remove an obsolete given or expected value column from the test specification.

Required inputs

There are no further user inputs required in order to perform a “Remove column” refactoring.

Input validation

Since no additional user input is required, this kind of refactoring does not perform any special input validation.

Workflow

The “Remove column” refactoring is carried out in the following sequence of steps:

(1) Remove the entire selected column from the test specification including all example values of this column.

(2) If the selected column is a given value column, carry out the following steps:

(2.1) Remove the corresponding public field from the *Fixture* class.

(2.2) Find all methods (including constructors) within the *Fixture* class that have a reference to this field. Comment the entire body of all found methods out. Additionally, add a TODO-comment and a “NotImplemented”-exception *throw* clause to each of those method bodies.

(3) If the selected column is an expected value column, carry out the following steps:

(3.1) Remove the corresponding public method from the *Fixture* class.

(3.2) Find all methods (including constructors) within the *Fixture* class that have a reference to the lately removed method. Comment the entire body of all found methods out. Additionally, add a TODO-comment and a “NotImplemented”-exception *throw* clause to each of those method bodies.

Example: Remove expected value column

The following example deals with a test specification intended to test the correct behaviour of a web hosting server. The fictional server blocks users who attempt to connect more than three times and allows access only for persons over 18 years of age and those who have not been blocked yet. Both the test specification and its corresponding *Fixture* can be seen in figure 25.

rule for	web access		
age	connection attempts	blocked?	admission?
14	1	false	false
22	1	false	true
23	4	true	false

```
1  public class WebAccess {
2      public int age;
3      public int connectionAttempts;
4
5      public bool Blocked() {
6          return (connectionAttempts > 3);
7      }
8
9      public bool Admission() {
10         return (age > 18 && !Blocked());
11     }
12 }
```

Figure 25: RuleFor test before "Remove expected column" refactoring

In this scenario, the expected value column named “blocked” is removed through the “Remove column” refactoring. The changes which were applied to the test specification as well as to the *Fixture* during the refactoring process are shown in figure 26.

rule for	web access	
age	connection attempts	admission?
14	1	false
22	1	true
23	4	false

```
1  public class WebAccess {
2      public int age;
3      public int connectionAttempts;
4
5      public bool Admission() {
6          // TODO: Check implementation
7          //return (age > 18 && !Blocked());
8          throw new System.NotImplementedException(
9              "This method needs to be implemented");
10     }
11 }
```

Figure 26: RuleFor test after "Remove expected column" refactoring

The “blocked” column was completely removed from the test definition as well as the

corresponding method in the *Fixture*. Since the “Admission” method references the deleted “Blocked” method, its body must be commented out to avoid compilation errors. A TODO comment and a “NotImplemented” exception *throw* clause are added to the body of the method in order to inform developers and customers about the modifications.

4.5.2.3 Rename Given / Expected Value Column

Motivation

With continuous software development, the context in which input parameters or methods conducting calculations are used, can change. This causes their names to become inaccurate or misleading. In such a case it is necessary to give them more reasonable names.

The “Rename column” refactoring can be used to easily change the name of a given or expected value column.

Required inputs

For the “Rename column” refactoring, the user must provide a new name for the given or expected value column.

Input validation

Similar to the “Add column” refactoring, the new name must be a valid C# identifier because it is mapped to a public field or a public method respectively within the *Fixture*. Moreover, it must be verified that the *Fixture* does not contain a field (in case of a given value column being renamed) or a method (in case of an expected value column being renamed) with the same name as the new column.

Workflow

The “Rename column” refactoring is carried out in the following sequence of steps:

(1) Rename the selected column by changing the column's header field to the new name.

(2) If the selected column represents given values, carry out the following steps:

(2.1) Rename the corresponding public field in the *Fixture* class.

(2.2) Find all references of this field and rename all field references, too.

(3) If the selected column represents expected values, carry out the following steps:

(3.1) Rename the corresponding public method in the *Fixture* class.

(3.2) Find all references of this method and rename all method references, too.

Example: Rename expected value column

The following example uses the same test specification and *Fixture* as in the antecedent example (see figure 27).

rule for	web access		
age	connection attempts	blocked?	admission?
14	1	false	false
22	1	false	true
23	4	true	false

```
1  public class WebAccess {
2      public int age;
3      public int connectionAttempts;
4
5      public bool Blocked() {
6          return (connectionAttempts > 3);
7      }
8
9      public bool Admission() {
10         return (age > 18 && !Blocked());
11     }
12 }
```

Figure 27: RuleFor test before "Rename expected column" refactoring

In this example, the “blocked” expected value column is renamed to “user is blocked” by applying the “Rename expected column” refactoring. Figure 28 shows the resulting test specification and *Fixture*.

rule for	web access		
age	connection attempts	user is blocked?	admission?
14	1	false	false
22	1	false	true
23	4	true	false

```
1  public class WebAccess {
2      public int age;
3      public int connectionAttempts;
4
5      public bool UserIsBlocked() {
6          return (connectionAttempts > 3);
7      }
8
9      public bool Admission() {
10         return (age > 18 && !UserIsBlocked());
11     }
12 }
```

Figure 28: RuleFor test after "Rename expected column" refactoring

The expected value column in the test specification was renamed as well as the corresponding method in the *Fixture*. The “Admission” method contained a reference of the renamed method (line 10), so this reference was renamed, too.

4.5.3 Refactorings For Scenario Tests

4.5.3.1 Add Action

Motivation

As explained in chapter 3.5.1.2, *Scenario* tests are used to test the dynamic behaviour of a system. They allow for specifying actions that are executed sequentially on the SUT.

When requirements change during development, it might be necessary to include new actions to the *Scenario* test to cover the new functionality. Whenever actions have to be added to an existing *Scenario* test, the “Add action” refactoring can be used.

Required inputs

The “Add column” refactoring requires five inputs from the user:

(1) Action text

The action which is to be executed on the SUT is expressed in natural language and must be specified by the user.

(2) Action type

There are five different action types (*Given*, *Then*, *When*, *Display*, and *Check*). They differ in the way the *GreenPepper* engine carries them out and in the way the engine handles the results of the execution of each action (see chapter 3.5.2.2). The action type also has an influence on the corresponding method signature and must therefore be specified by the user.

(3) Regular expression

Since each *Scenario* action is mapped to a method through a regular expression (see chapter 3.5.2.2), a regular expression must also be provided by the user.

(4) Method name

Although the name of the corresponding method in the *Fixture* does not play any important role for the refactoring of *Scenario* tests, the user should provide a reasonable method name to keep the *Fixture* as clear and comprehensible as possible.

(5) Insertion position

Since the order in which the *Scenario* actions are executed can have an influence on the result returned by the SUT, the user must specify the position at which the new action shall be inserted within the test specification.

Input validation

Following restrictions apply to the input data provided by the user for the refactoring to succeed:

- The action text must match the provided regular expression, so that the mapping process from the *Scenario* action to the corresponding method in the *Fixture* can be conducted successfully after applying the refactoring.
- The method name must be a valid C# identifier.

- The insertion position must be within the valid range.

Workflow

The “Add action” refactoring is conducted as follows:

(1) Add a new action to the *Scenario* test specification at the specified position and with the provided action text.

(2) If the corresponding *Fixture* class contains a method that already “matches” the new action text, do not apply any changes to the *Fixture*. In that case, ignore the provided action type, regular expression and method name.

(3) If the corresponding *Fixture* class does not contain a “matching” method, conduct the following steps:

(3.1) Add a public method with the specified name to the *Fixture* class.

(3.2) Adjust the method signature depending on the action type selected by the user (see table 3 of chapter 3.5.2.2).

(3.3) Search the regular expression for grouping constructs (see chapter 3.5.2.2) and add an “object”-typed parameter to the method's parameter list for each grouping construct.

(3.4) Annotate the newly created method with the appropriate *GreenPepper* attribute and the provided regular expression.

Example: Add scenario action

Figure 29 shows an example of a *Scenario* test for a simple fictional bank application. The test contains two actions that are to verify that there is no balance in a newly opened bank account.

scenario	Bank
open account 12345 under the name of Denis Elbert	
verify that balance of account 12345 is $0	

```
1  public class Bank {
2
3      [Given("open account (\\d{5}) under the name of ([\\w|\\s]*)")]
4      public void OpenAccount(string accountNr, string name) { ... }
5
6      [Then("verify that balance of account (\\d{5}) is \\$(\\d+)")]
7      public void BalanceOfAccount(string accountNr,
8          Expectation expectedBalance) { ...}
9  }
```

Figure 29: Scenario test before "Add action" refactoring

As can be seen in the *Fixture* below the test specification, each action is mapped to a method that is annotated with a regular expression that matches one of these action texts.

When the customer decides to add more actions to the *Scenario* test, he can do that through the “Add action” refactoring. As an example, it is assumed that the customer adds another action for depositing a hundred dollars to the newly opened bank account and that he provides following input data:

- The action is describes by the text “deposit $100 in account 12345”.
- Since the action is assumed to bring the SUT in another state, the *When* action type is selected.
- The regular expression “deposit \$(\d+) in account (\d{5})” is provided in order to match a random amount of money and any five digit account number.
- The method name is chosen to be “Deposit”.

Figure 30 shows the resulting changes to the test specification and its corresponding *Fixture* that were applied by the “Add action” refactoring based on the provided input data.

scenario	Bank
open account 12345 under the name of Denis Elbert	
verify that balance of account 12345 is $0	
deposit $100 in account 12345	

```
1  public class Bank {
2
3      [Given("open account (\\d{5}) under the name of ([\\w|\\s]*)")]
4      public void OpenAccount(string accountNr, string name) { ... }
5
6      [Then("verify that balance of account (\\d{5}) is \\$(\\d+)")]
7      public void BalanceOfAccount(string accountNr,
8          Expectation expectedBalance) { ...}
9
10     [When("deposit \\$(\\d+) in account (\\d{5})")]
11     public void Deposit(object param1, object param2) {
12         // TODO: Implement test method, edit regular expression and edit method parameter list
13         throw new System.NotImplementedException(
14             "This method needs to be implemented");
15     }
16 }
```

Figure 30: Scenario test after "Add action" refactoring

Since the user provided regular expression contained two grouping constructs, two method parameters (*param1* and *param2*) were automatically generated for the new method.

4.5.3.2 Remove Action

Motivation

For the same reason that actions have to be added to a *Scenario* test, they need to be removed in certain circumstances, too. A good example in this context is when system features are discarded due to changed requirements. Hence, actions that are related to these features must be removed from the test specification.

In order to remove actions from a *Scenario* acceptance test, the “Remove action” refactoring can be applied.

Required inputs

The “Remove action” refactoring does not require any input data.

Input validation

Since no additional user input is required, this kind of refactoring does not perform any special input validation.

Workflow

The “Remove action” refactoring is conducted as follows:

(1) Remove the selected action from the *Scenario* test specification.

(2) Find all methods in the corresponding *Fixture* that match the selected action. For each of those methods carry out step 3.

(3) Find all remaining *Scenario* actions from the (modified!) test specification that match the current method. If no action could be found, execute the following steps with the current method:

(3.1) Delete the current method from the *Fixture* class

(3.2) Find all methods (including constructors) within the *Fixture* class that have an reference to the lately deleted method. Comment the entire body of all found methods out. Additionally, add a TODO-comment and a “NotImplemented”-exception *throw* clause to each of those method bodies.

Example

The previous chapter introduced an example of a *Scenario* test specification for a fictional bank application. In this example a new *Scenario* action was added through the “Add action” refactoring. Figure 30 shows the changes that were applied to the test specification and the corresponding *Fixture* in the course of this refactoring.

When the “Remove action” refactoring is applied to the lastly added action, the original state shown in figure 29 is restored. The refactoring deletes the action from the test specification as well as the corresponding method in the *Fixture*.

4.5.3.3 Edit / Rename Action

Motivation

During the ongoing development of a system, it can happen that the nomenclature changes after the acceptance tests have been written. Furthermore, *Scenario* actions are written in natural language, so they are especially prone to typing errors. In order to keep the test specification consistent and comprehensible it is necessary to rename the *Scenario* actions. To do so, the “Rename action” refactoring can be used.

Required inputs

The new action text is required for the “Rename action” refactoring along with its associated regular expression.

Input validation

The original action text contains sections that are interpreted as parameters during the mapping process of the *Scenario* action to the corresponding *Fixture* method (see chapter 3.5.2.2). These sections are identified with the help of the regular expression associated with this action. More precisely, the grouping constructs within the regular expression distinguish the parameters from the descriptive text.

However, only the descriptive text is allowed to be modified and the sections representing parameters must remain the same. Moreover, it must be ensured that the new action text matches its associated regular expression.

Workflow

The “Rename action” refactoring is executed in the following steps:

(1) Rename the selected *Scenario* action to the specified new text.

(2) Find the method within the corresponding *Fixture* class that matches the old action name.

(3) For each Scenario action that matches this method, rename the current action to the new text.

(4) Change the regular expression associated with this method to the new regular

expression.

Example

Below, the bank example from the previous chapters is used again to demonstrate the "Rename action" refactoring. The exemplary test specification defines actions to verify that the system computes the balance of a bank account correctly after opening and depositing money in the account. The *Scenario* acceptance test and the corresponding *Fixture* can be seen in figure 31.

scenario	bank
open account 12345 under the name of Denis Elbert	
verify that balance of account 12345 is $0	
deposit $100 in account 12345	
verify that balance of account 12345 is $100	

```
1  public class Bank {
2
3      [Given("open account (\\d{5}) under the name of ([\\w|\\s]*)")]
4      public void OpenAccount(string accountNr, string name) { ... }
5
6      [When("deposit \\$(\\d+) in account (\\d{5})")]
7      public void Deposit(double amount, string accountNr) { ... }
8
9      [Then("verify that balance of account (\\d{5}) is \\$(\\d+)")]
10     public void BalanceOfAccount(string accountNr,
11         Expectation expectedBalance) { ...}
12 }
```

Figure 31: Scenario test before "Rename action" refactoring

In this example, the "verify"-action located in the third line of the test shall be renamed to "verify that *current* balance of account 12345 is $0" in order to emphasize that the *current* balance of the account is being tested by this action. Therefore, the "Rename action" refactoring is applied. The results of the refactoring are shown in figure 32.

scenario	bank
open account 12345 under the name of Denis Elbert	
verify that current balance of account 12345 is $0	
deposit $100 in account 12345	
verify that current balance of account 12345 is $100	

```
1  public class Bank {
2
3      [Given("open account (\\d{5}) under the name of ([\\w|\\s]*)")]
4      public void OpenAccount(string accountNr, string name) { ... }
5
6      [When("deposit \\$(\\d+) in account (\\d{5})")]
7      public void Deposit(double amount, string accountNr) { ... }
8
9      [Then("verify that current balance of account (\\d{5}) is \\$(\\d+)")]
10     public void BalanceOfAccount(string accountNr,
11         Expectation expectedBalance) { ...}
12 }
```

Figure 32: Scenario test after "Rename action" refactoring

As can be seen above, both “verify”-actions of the test specification have been affected by the refactoring, and renamed appropriately.

In the *Fixture* class, the “BalanceOfAccount” method was mapped to the “verify”-action because the action matched the regular expression associated with the method. As a result of the refactoring mechanism, this regular expression was changed in such a way as to match the new “verify”-action text.

4.6 Graphical User Interface (GUI)

As explained in the previous chapters, each refactoring requires user inputs in order to be carried out. The “Rename test” refactoring, for example, requires the user to enter a new name. Thus, the refactoring extension developed as part of this thesis must provide a graphical user interface (GUI) that enables the user to enter the required information.

The goals of this thesis stipulate that the refactoring extension must be implemented for the latest version of *Visual Studio,* which is *Visual Studio 2010*. One of the major differences between this version and older versions of *Visual Studio* is that its entire graphical user interface was rewritten using the *Windows Presentation Foundation* (WPF) framework (see

chapter 3.6.3). That is why WPF was used for the implementation of the refactoring GUI, too.

The following list discloses all GUI elements that were needed for the refactoring extension:

- “Rename test” refactoring dialogue
- “Add column” refactoring dialogue
- “Rename column” refactoring dialogue
- “Add action” refactoring dialogue
- “Rename action” refactoring dialogue
- Preview dialogue

Chapter 5.4 explains how these GUI elements were implemented using WPF.

5 Implementation

5.1 GreenPepper Acceptance Test Parser

This chapter explains the design of the *GreenPepper* acceptance test parser that is needed to access the information stored in the test specification and to modify this information whenever a test refactoring is carried out.

As figure 33 shows, the *GreenPepper* acceptance test parser is subdivided into three layers. Each layer has its own sub-parser and produces an object-model based on its input data. The created object-model wraps the information that is extracted by the respective sub-parser and serves as input for the overlying layer.

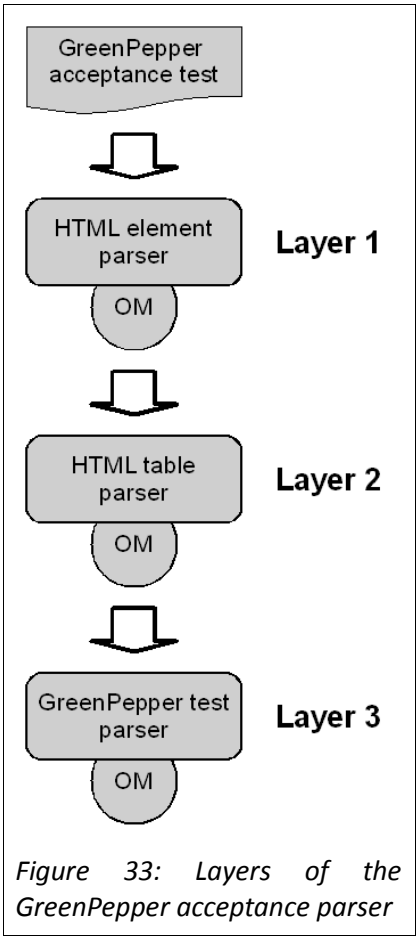


Figure 33: Layers of the GreenPepper acceptance parser

In the following chapters, the purpose and design of each layer is explained in greater detail.

5.1.1 Parser Layer 1

The first layer expects a *GreenPepper* acceptance test file - which is a HTML file - as input. The purpose of this layer is to discover all HTML tags that are contained in the file and to provide an object-model that allows for accessing the identified tags in the same order as they appear within the file.

HTML elements

The sub-parser which is responsible for creating the object-model distinguishes between three HTML elements and represents each of them by an own class in the object-model. In order to identify the distinctive elements within the HTML file, the sub-parser uses regular expressions. The table below gives an overview of all three elements and shows the specified regular expression used to identify each of them.

Element	Interpretation	Regular expression
Comment	Represents text that is specified between HTML comment tags. Comment tags start with “<!--” and end with “-->”.	`\<\!--([^-]|-[^-]|--[^>])*--\>`
Tag	Represents all tags that are no comment tags. In general, HTML tags start with “<” and end with”>”.	`</?\w+((\s+\w+(\s*=\s*(?:\".*?\"|'.*?'|[^'\">\s]+))?)+\s*|\s*)/?>`
Other	Represents all file content that is neither a comment nor another HTML tag.	No regular expression is needed. The “*Other”* element is identified when none of the above regular expression matched.

Table 6: Element types identified by the first level parser

Different types of tags

Since *GreenPepper* acceptance tests are expressed in HTML tables, the generated object-model must support to check what kind of tag is represented by a “*Tag*”-element, especially table tags such as *<table>, <tr>, <th>,* and *<td>*. Therefore, regular expressions were also used (see table 7).

Type of tag	HTML tag	Regular expression
Table	<table>	</?table(\s.*|/?>)
Table row	<tr>	</?td(\s.*|/?>)
Table cell	<td>	</?td(\s.*|/?>)
Table header cell	<th>	</?th(\s.*|/?>)

Table 7: Identification of different types of HTML tags

Object-model design

Regarding the implementation of the object-model, all three HTML element classes are derived from a common base class named *TextElement* as can be seen in the class diagram of figure 34. The diagram shows the entire structure of the first parser layer.

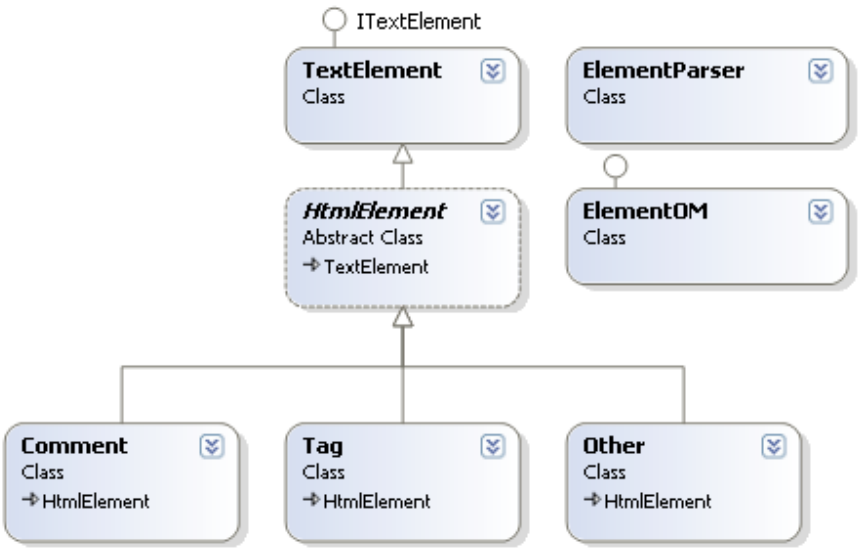


Figure 34: Class diagram for the first parser layer

The *TextElement* class implements basic properties which can be accessed through the *ITextElement* interface. These properties are used to save the positional information of each HTML element within the file as well as the represented text itself. More precisely, the exact character index of where the corresponding text part begins and ends in the file is stored in these properties.

The *ElementParser* class represents the actual sub-parser and creates the object-model for this layer. It therefor parses through the input file by applying the regular expressions and adds the

identified HTML elements sequentially to a list in the object-model. In doing so, the parser appends the position data to each HTML element.

5.1.2 Parser Layer 2

The second layer is responsible for identifying HTML tables and their logical structure, based on the list of HTML tags identified by the first layer's parser. The object-model created by layer one is therefore used as an input for this layer, whose output is an object-model reflecting the logical structure and content of HTML tables.

Table representation

Figure 35 shows the internal structure of the second parser layer.

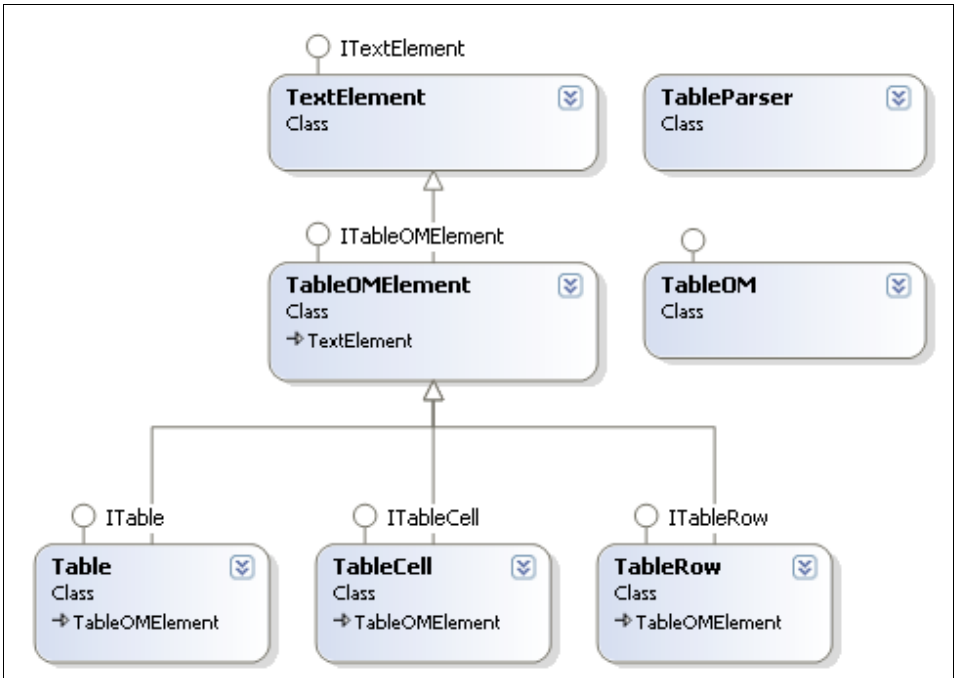


Figure 35: Class diagram for the second parser layer

As can be seen in the class diagram, three classes were introduced to reflect the logical structure of a HTML table. The classes are linked to each other using the concept of object composition and are briefly described below:

- **Table**

The *Table*-class represents an entire HTML table, i.e. all text that is noted down between a starting end an ending table tag (*<table>* and *</table>*). It contains a list of *TableRow* objects (object composition).

- **TableRow**

  The *TableRow*-class represents one single table row, i.e. all text that is noted down between the respective starting and an ending table row tag (*<tr>* and *</tr>*). The class contains a list of *TableCell* objects (object composition) and a reference to the parental *Table* object.

- **TableCell**

  The *TableCell*-class represents one single table cell, i.e. all text that is noted down between the respective starting and an end ending table cell tag (*<td>* and *</td>*, *<th>* and *</th>*). It contains a reference to the parental *TableRow* object.

Ignored data

One requirement of the test specification parser was to preserve the content and format of the input file. Therefore, it was not only necessary to save the table-related data, but also to include the remaining content into the object model such as HTML comments or untagged text.

To achieve that, each of the three classes described above also contains a list so called *Ignored Data* list. During the parsing process, all "*Comment"*- and "*Other"*-elements as well as *"Tag"*-elements not representing table tags are assigned to that list.

Parsing

The objective of layer two is to generate an object model that contains a list of *Table*-objects and abstracts from the actual structure of the HTML file. As mentioned before, the layer two sub-parser parses through the output from layer one in order to generate the object model.

The sub-parser therefor operates in four major steps, which are explained below. In each of those steps, the input list of HTML elements (the list belongs to the output of the layer one parser) is traversed whereas elements that have already been processed are disregarded.

(1) Generate object model

(1.1) Add all HTML elements to the *Ignored Data* list of the object model until a starting table tag is discovered.

(1.2) Check if a table can be retrieved by searching for an ending table tag. If so, execute step 2 and add the table to the object model. If not, add the starting table tag to the *Ignored Data* list of the object model.

(1.3) Repeat steps 1.1 to 1.2 until all HTML elements have been processed. Return the object model.

(2) Gather table

(2.1) Add all HTML elements to the *Ignored Data* list of the table object until a starting row tag is discovered.

(2.2) Check if a row can be retrieved by searching for an ending row tag. If so, execute step 3 and add the row to the table object. If not, add the starting row tag to the *Ignored Data* list of the table object.

(2.3) Repeat steps 2.1 to 2.2 until all HTML elements have been processed. Return the table object.

(3) Gather row

Execute steps accordingly to (2).

(4) Gather cell

(4.1) Assign all HTML elements as content to the cell until the ending cell tag is discovered.

(4.2) Return the cell object.

5.1.3 Parser Layer 3

As described in the previous chapter, the second parser layer generates an object model that

allows for easily accessing tables including containing rows and cells which are specified within a *GreenPepper* acceptance test file. The purpose of the third layer is to abstract from these tables and to provide an object model for accessing and modifying the information specified in *GreenPepper* tests.

GreenPepper test interfaces

The figure below shows the definition of the object model interfaces for both the *RuleFor*- and the *Scenario* acceptance test. The interfaces define properties used to access particular test parts (e.g. the list of actions in case of a *Scenario* test) and methods used to modify the tests (e.g. to add an action to a *Scenario* test).

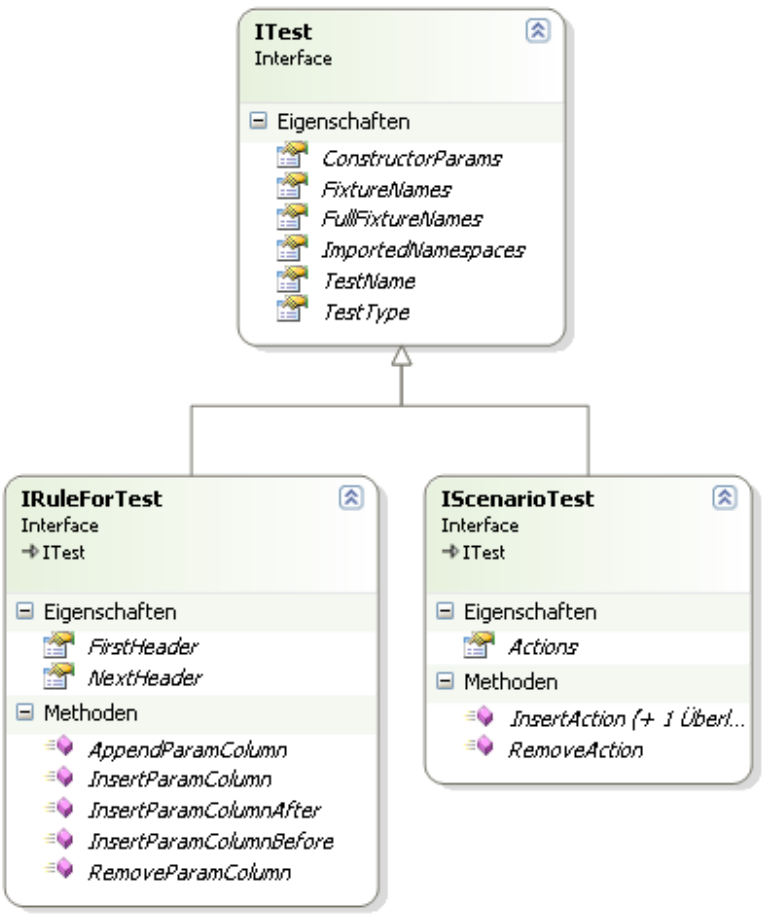


Figure 36: Interfaces of layer 3 representing GreenPepper acceptance tests

In the following, some selective properties are described:

- **FixtureNames**

  As explained in chapter 3.5.2, *GreenPepper* test names are linked to the corresponding *Fixture* class name. Since the mapping can be ambiguous (e.g. the test name “bank” can both be mapped to “Bank” and “BankFixture”), this property provides a list of all possible *Fixture* names.

- **ImportedNamespaces**

  If an *Import* interpreter (see chapters 3.5.2 and 3.5.1.3) is specified before a test, namespaces have to be included in the test name mapping process. This property provides a list of all imported namespaces.

- **FullFixtureNames**

  This property provides a list of all possible full qualified *Fixture* names by combining the list of namespaces with the *Fixture* names.

Next to the major test interfaces there are two special interfaces that are implemented by several special classes: The *IRenameable* interface and the *ILocatable* interface. Both are explained below in greater detail.

IRenameable interface

There are three *GreenPepper* acceptance test elements that can be renamed: Test names, expected and given *RuleFor* columns, and *Scenario* actions. The *IRenameable* interface is implemented by the respective classes that represent these *GreenPepper* parts. As can be seen in figure 37, the interface provides two properties and one method.

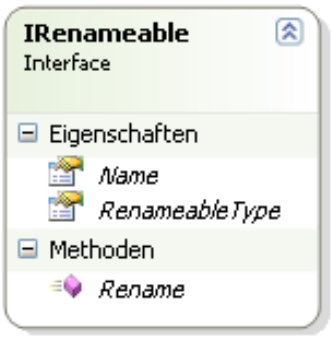


Figure 37: IRenameable interface

The *Name* property simply returns the name of whatever element needs to be renamed, whereas the *Rename* method can be used to change the name. The *RenameableType* property specifies the current *GreenPepper* element which is to be renamed.

Whenever new elements that need to be renamed have to be included to the object model, the *IRenameable* interface can be implemented to support that functionality.

ILocatable interface

As mentioned in chapter 4.2, a refactoring is initiated by right-clicking on the particular acceptance test part intended to being refactored in the editor of *Visual Studio*. After the right-click position within the test file has been retrieved (see chapter 5.3), the *GreenPepper* acceptance test element that corresponds to this location must be identified automatically in order to carry out the respective refactoring actions.

For this purpose, the *ILocatable* interface was introduced (see figure 38) which is implemented by each class representing a *GreenPepper* test element that needs to be located. It provides two methods which are described below:

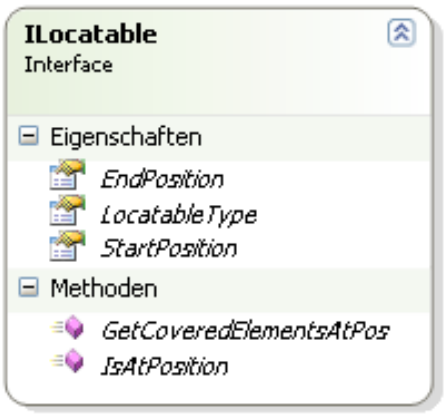


Figure 38: ILocatable interface

- **"IsAtPosition" method**

  This method returns *true* if the current object model element is located at the specified character offset. It therefor uses the start and end position data from the *ITextElement* interface mentioned earlier that comes with every object model element.

- **"GetCoveredElementsAtPos" method**

  This method returns all child elements that are located at the specified character offset. In order to identify these, the method itself calls the *IsAtPosition* method of all child elements that implement the *ILocatable* interface.

5.1.4 Parser Usage

The previous chapters explained the design of each of the three test parser layers. The highest abstraction level is achieved by the last layer – layer three – which creates an object model that can be used to access and modify the information within a *GreenPepper* acceptance test.

In order to get a reference to that object model within the refactoring extension application, the sub-parsers corresponding to each layer have to be called sequentially, passing the results to the next higher layer. Figure 39 demonstrates the code necessary to get a test object model reference. It is assumed, that the path to the *GreenPepper* acceptance test file that is be parsed was assigned to the "*documentPath*" variable in precursory steps.

```
{
    ...

    IElementOM om1 = new ElementParser(documentPath).Parse();
    ITableOM om2 = new TableParser(om1).Parse();
    ITestOM testObjectModel = new TestParser(om2).Parse();

    ... //use test object model
}
```

Figure 39: Code for retrieving the test object model

5.2 Refactoring Commands

As mentioned in chapter 4.1, the decision was made to place the refactoring commands in the context menu of the design and code view of the *Visual Studio* IDE. There are a total of seven refactoring commands that had to be added to the context menu:

- Rename test command
- Add / Rename / Remove column command
- Add / Rename / Remove action command

Context menu layout

In order to incorporate the new refactoring commands into the context menu, the VSCT-file (see chapter 3.6.2.2), which came along with the *GreenPepe2010 VSPackage,* was utilized.

First of all, the different commands were defined in the *Commands* section of the VSCT-file. Figure 40 shows how this was done using the example of the “Rename” command.

```
<Button guid="guidGPRefactoringCmdSet"
        id="cmdRename" type="Button">

    <Icon guid="guidIcons" id="edit"/>
    <CommandFlag>DynamicVisibility</CommandFlag>
    <CommandFlag>TextChanges</CommandFlag>
    <Strings>
      <ButtonText>Rename</ButtonText>
    </Strings>
</Button>
```

Figure 40: Definition of the "Rename" command in the VSCT file

As can be seen in the figure above, all major properties (icon, text, type) of a command can be specified within the VSCT file. The ID is used to identify each defined command throughout the VSCT file. With the help of that ID, the relationship between other command elements such as groups and menus can be configured within the *CommandPlacements* section of the VSCT file.

In order to place commands in the context menu of the *Visual Studio* editor, the IDs of the respective context menus - which are defined as constants in the Visual Studio SDK – had to be assigned to the custom commands. Figure 41 shows the respective code to reference these IDs (“49” and “51”).

```
<GuidSymbol name="guidRightClickCmdSet"
            value="{D7E8C5E1-BDB8-11D0-9C88-0000F8040A53}">
    <IDSymbol name="cmdRightClick1" value="51"/>
    <IDSymbol name="cmdRightClick2" value="49"/>
</GuidSymbol>
```

Figure 41: Access to the VS context menu

Dynamic behaviour

Based on what kind of acceptance test (*Scenario* test or *RuleFor* test) was selected during a right click in the design or code view, only a subset of all commands must be displayed. For example, if the cursor is positioned over the name of a *Scenario* acceptance test when right-clicking, the context menu should contain only the “Rename test” command and “Add action” command.

In order to achieve this behaviour, the visibility of the commands must be manipulated during

runtime. Therefore, two special *CommandFlags* have to be defined in the *Commands* section for each command: *DynamicVisibility* and *TextChanges* (see figure 40). Once these flags are defined, a command can be represented in code by an *OleMenuCommand* object, which is provided as part of the *Visual Studio* SDK. This object offers two properties *Enabled* and *Visible,* which can simply be set to either *true* or *false* in order to activate and deactivate or to show and hide a command.

Figure 42 shows an example of the final implementation of the context menu that popped up after the user right-clicked on the test name of a *Scenario* test.

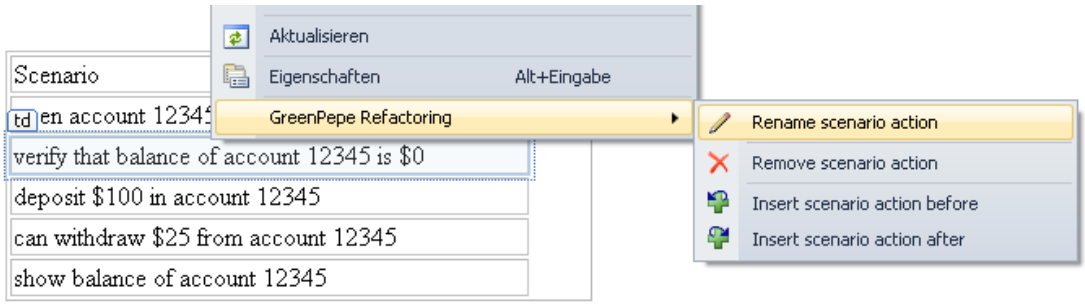


Figure 42: Example of the refactoring context menu

5.3 Document Interaction In Visual Studio

As explained in chapter 4.2, it was necessary to determine the position - or more precisely the exact character offset – where the user right-clicked on in the test specification file.

The DTE (see chapter 3.6.2.1) object, which comes with the *Visual Studio* SDK, provides a property named *ActiveDocument* that represents the currently opened document in the *Visual Studio* IDE. Figure 43 shows how to obtain a reference to this object.

```
1  {
2      ...
3
4      DTE dte = (DTE) vsPackage.GetService(typeof(SDTE));
5      Document doc = dte.ActiveDocument;
6
7      ...
8  }
```

Figure 43: Code to retrieve the opened document in VS

The *Document* object allows for accessing all relevant information about the opened document such as the file name, the full file path, the status whether the document is write protected and more. It also contains a property called *Selection* which – amongst others - holds information about the current cursor position within the document. Since the cursor is automatically set to the position where the user right-clicked, this property could be used to determine the required absolute character offset within the document.

5.4 Graphical User Interface (GUI)

As mentioned earlier, all graphical user interface (GUI) elements were implemented using WPF (see chapter 3.6.3).

One of the big advantages of WPF is the potential to separate the design specification from the logic implementation. This was done by specifying all design related information in XAML-files. The complete layout for the GUI including all required buttons, text boxes, combo boxes, check boxes and so on was specified using XAML.

The logic was put into so called *User Controls*. These are re-usable graphical components that can be included and re-used on other controls just like a text box can be positioned on a custom user control. The user controls were implemented by deriving from the *UserControl* class which is part of the WPF library.

Since user controls cannot be displayed by themselves, each user control was put in a *Window* container class. The *Window* class is the base class for creating and displaying custom windows.

In the following, a subset of the GUI elements that were created in the course of this work are

described. The input controls provided in each implemented window are closely related to the inputs that were considered to be essential in the course of chapter 4.5.

"Add column" dialogue

Figure 44 shows the window that is displayed whenever an "Add column" refactoring is executed.

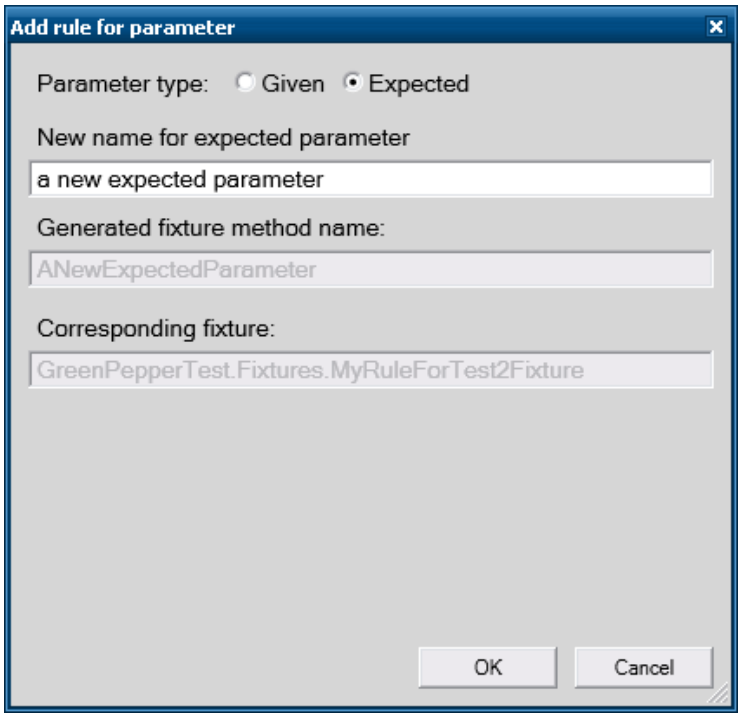


Figure 44: GUI for "Add column" refactoring

As can be seen in the screenshot of the "Add column" window, it is possible to select whether a given or expected column shall be added. Depending on the selection and the provided column name, the GUI control automatically generates the corresponding *Fixture* method name (in case of an expected column) or field name (in case of a given column) which is linked to the column name. The GUI control also displays the *Fixture* class name that is connected with the respective acceptance test.

"Add Scenario action" dialogue

Whenever an "Add action" refactoring is performed, the following window will be displayed (see figure 45):

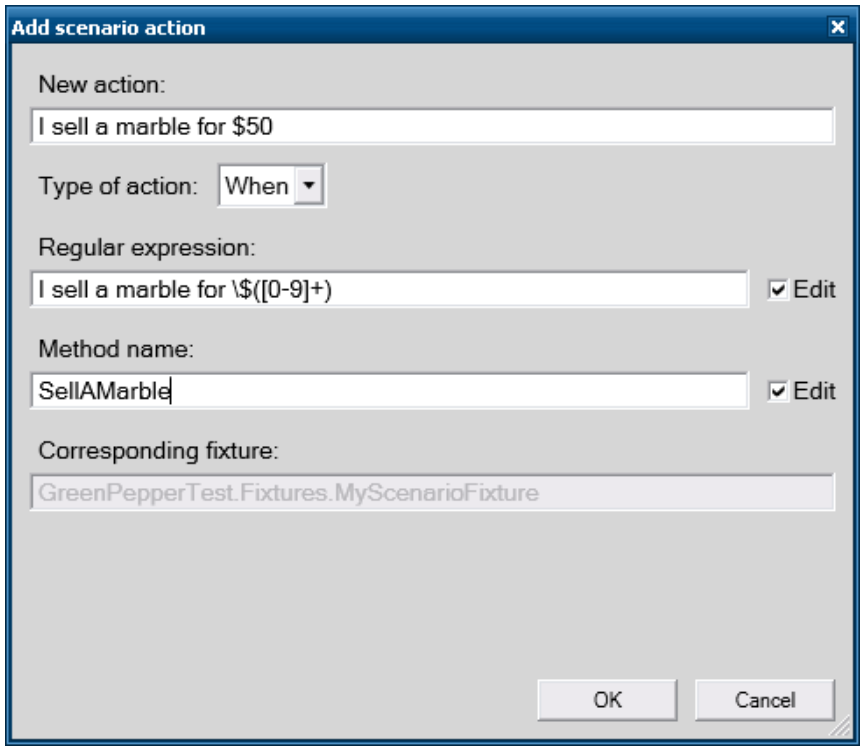


Figure 45: GUI for the "Add action" refactoring

This dialogue allows for entering the new *Scenario* action name and the selection of the *Scenario* action type (e.g. "*When"*).

Based on the provided action name, the GUI control automatically generates a regular expression, which can be edited manually if necessary. If the action name does not match the regular expression, the GUI controls displays an error message and deactivates the OK-button until the error has been corrected by the user.

The method name is also generated automatically, but can be edited by the user. The typed method name must be a valid C# identifier, otherwise an error message is shown.

"Rename Scenario action" dialogue

Figure 46 shows the typical window for a "Rename action" refactoring.

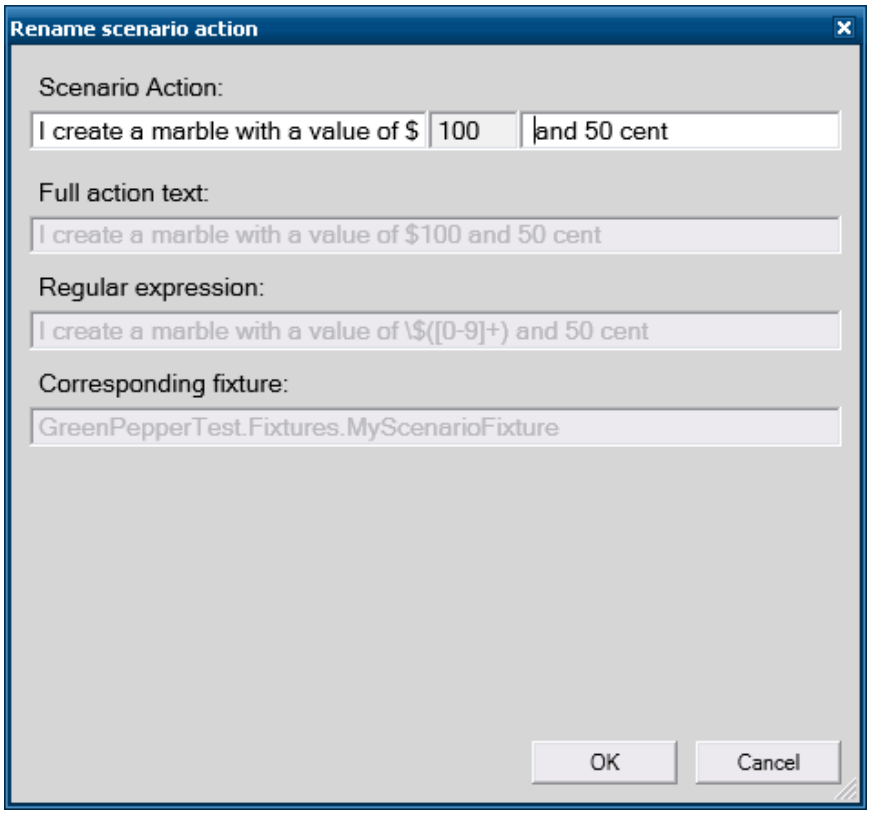


Figure 46: GUI for the "Rename action" refactoring

Based on the regular expression connected to the *Scenario* action that is to be renamed, the original *Scenario* action text is split into different parts. This splitting algorithm sets the *Scenario* action parameters apart from the descriptive text surrounding them (see chapter 4.5.3.3). Only the “descriptive text” parts are allowed to be edited. Based on the user input, the GUI controls generates the regular expression linked to the new action name.

“Preview” dialogue

The “Preview” dialogue, which is displayed before the changes of a refactoring are applied to the test specification and the corresponding *Fixture*, can be seen in figure 47.

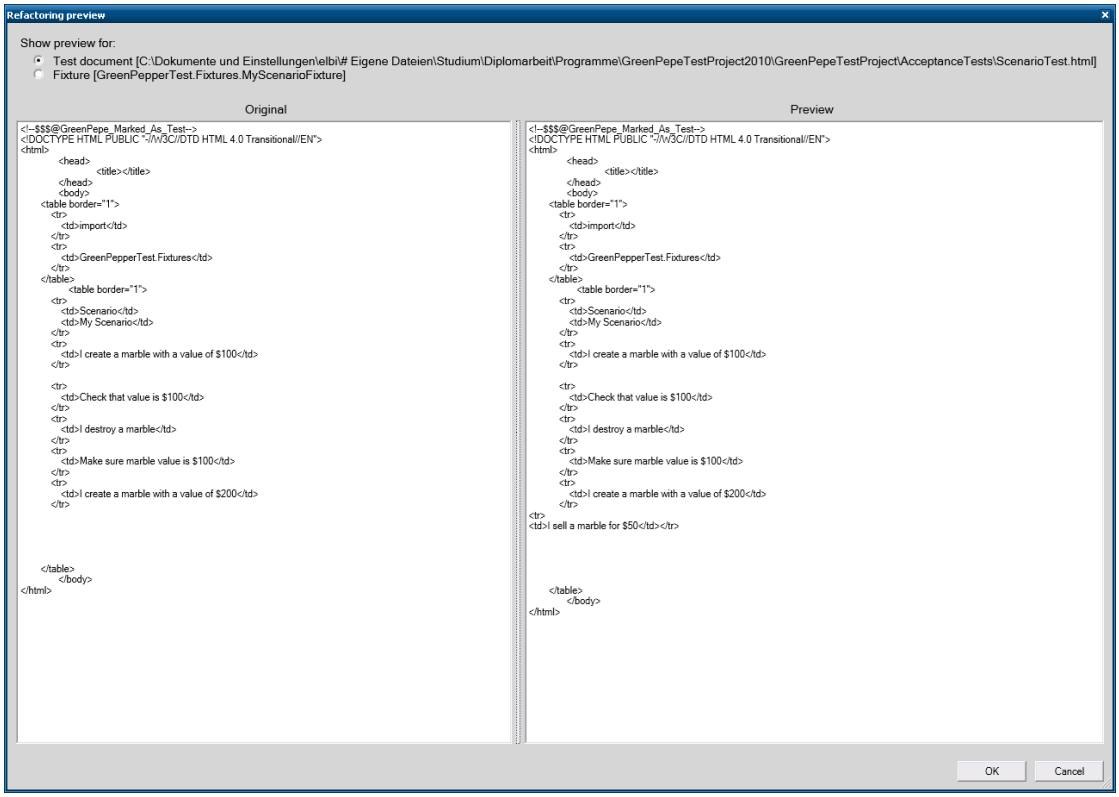


Figure 47: Preview dialogue

In the current implementation, the “Preview” dialogue simply displays the content of either the test specification or the Fixture before (left side) or after (right side) a particular refactoring. So far, no syntax highlighting is supported, but this could be incorporated in future time.

5.5 Core Classes And Interfaces

This chapter explains some of the major interfaces and classes that build the foundation pillar of the refactoring extension application implemented in the course of this work.

5.5.1 IClassCodeManipulator Interface

As mentioned earlier in this thesis, the refactoring of acceptance tests requires to manipulate the source code of the *Fixtures*. Therefore, the *IClassCodeManipulator* interface was created. It defines all methods that are needed to retrieve information from a *Fixture* class as well as to

modify it.

Amongst others, the interface includes methods to ...

- check for defined fields, methods and constructors within a class.
- remove public fields and methods from a class.
- rename fields, methods and constructors within a class including their references.

Through the definition and usage of this interface throughout the refactoring application whenever the code of a *Fixture* needed to be accessed, it was possible to abstract from the current implementation of the source code parser. Chapter 4.4 gives reasons for the usage of a third party library called *NRefactory*. This library comes along with a C# source code parser that was used to implement the *IClassCodeManipulator* interface. However, whenever another C# parser must be used, it can easily be exchanged by creating a new implementation of the *IClassCodeManipulator* interface.

5.5.2 Fixture Class

As the name may already imply, the *Fixture* class is the runtime representation of the actual *Fixture* that is linked to an acceptance test during the refactoring process. It specifies methods that are called by the *RefactoringExecuter* object (see chapter 5.5.4) in order to make all necessary modifications to the class connected to a *Fixture*. The modifications are performed by calling the methods of the *IClassCodeManipulator* interface which is described in chapter 5.5.1.

Beyond that, calling the methods of this *Fixture* class does not have an immediate impact on the actual *Fixture* file. Instead, all actions are performed internally within the *Fixture* representation object without modifying the original *Fixture* source code file. This makes it possible to provide a preview of the changes that a refactoring action would cause. In conjunction with this preview ability, the *Fixture* class provides two properties and two methods which are described below:

- **“ApplyChanges()“ method**

The *ApplyChanges()* method is called to apply all modifications to the actual original class file connected to a *Fixture*.

- **"DiscardChanges()" method**

  The *DiscardChanges()* method will cause the *Fixture* class to discard all lately performed modifications and to reload the actual *Fixture* class file.

- **"Preview" property**

  The *Preview* property returns the modified *Fixture* class source that resulted from calling the *Fixture* editing methods. It contains all changes that have been applied internally after the last call of the *ApplyChanges()* method.

- **"Code" property**

  The *Code* property returns the source code of the actual original *Fixture* class file. It represents the original status of the *Fixture* class before changes had been applied to it using the *Fixture* representation class.

5.5.3 TestDocument Class

The *TestDocument* class is similar to the *Fixture* class described above. Instead of representing a *Fixture* that is connected to a *GreenPepper* acceptance test, it represents the test specification itself. It also specifies methods that are called by an instance of the *RefactoringExecuter* class. The methods reflect all actions that can possibly be performed on a *GreenPepper* acceptance test in the course of a refactoring. More precisely, the specified methods are used to manipulate a test specification.

Also similar to the *Fixture* class, the *TestDocument* class does not apply the changes immediately to the actual acceptance test file, but possesses the same preview abilities as the *Fixture* class (see chapter 5.5.2). It therefor also offers the *ApplyChanges()* and *DiscardChanges()* methods as well as the *Preview* property. Its specified *Content* property is related to the *Code* property and is used to retrieve the original test specification before any changes have been applied to the acceptance test.

Beyond that, a *TestDocument* object instance is connected to the currently opened document

in the *Visual Studio* editor. In other words, it always represents that particular acceptance test file, which is currently opened in *Visual Studio* and which the user right-clicked on in order to invoke a refactoring command. How the connection was established is explained in greater detail in chapter 5.3.

5.5.4 RefactoringExecuter Class

The *RefactoringExecuter* class is the core element of the refactoring extension application that combines all the functionality together. It implements the refactoring workflow described in chapter 4.2 by carrying out the required steps for each kind of refactoring.

In the following, the general sequence of actions that is carried out by the *RefactoringExecuter* class is described:

(1) Obtain a reference to the *TestDocument* object, which represents the *GreenPepper* acceptance test that is currently opened in the editor of *Visual Studio* (see chapter 5.5.3).

(2) Obtain a reference to the Fixture object, which represents the *Fixture* connected to the *GreenPepper* acceptance test (see chapter 5.5.2).

(3) Initialize and show the graphical user interface depending on what kind of refactoring command was selected to retrieve the input data from the user.

(4) Call the methods of the *TestDocument* object to modify the test specification appropriate to the kind of refactoring that was selected.

(5) Call the methods of the *Fixture* object to modify the Fixture, based on what kind of refactoring was selected.

(6) Initialize and show the preview dialogue, which displays all changes that will be performed in the course of the selected kind of refactoring.

(7) Apply the changes by using the *TestDocument* and *Fixture* objects if the user confirmed the changes in the preview dialogue.

The *RefactoringExecuter* class is instantiated as soon as a refactoring command was selected by the user in the context menu of the *Visual Studio* editor.

5.5.5 ISettings Interface

Like most other applications, the refactoring extension implemented in the course of this work involves a collection of settings that have an influence on certain functionality of the system. For example, it comprises a setting to adjust the TODO-comment string that is added to the body of a newly generated method when a particular refactoring is executed.

All settings of the refactoring extension are specified at a central place within one class, the *Settings* class. An external storage location – a configuration file for example – was not required at that time. However, since it is possible that further development will require to manage the settings differently, the *ISettings* interface was introduced. All application settings are loaded through this interface. Whenever the settings need to be stored in a different way, this interface can be implemented to incorporate the changes.

6 Conclusion And Future Work

6.1 Problems

Several problems arose in the course of this work and are described below:

- **C# parser**

  As explained in chapter 4.4, a third party C# parser was needed in order to be able to parse the *Fixture* code. It was very time consuming to find an appropriate parser that met all the requirements and that made it possible to implement the refactoring functionality.

- **Beta versions**

  One of the goals of this thesis was to develop the refactoring extension for the latest version of the *Visual Studio* IDE. At that time, the first beta version of *Visual Studio* 2010 (VS 2010) was released including the new .NET version 4.0. Thus, the development of the refactoring extension took place using the VS 2010 IDE beta version. When bugs occurred during the development, it was sometimes hard to find out whether these were caused by the own implementation or by a bug within the IDE.

6.2 Summary And Evaluation

As described in the motivation part of this thesis, there has not been acceptance test refactoring support for the .NET environment yet.

In the course of this work, a *Visual Studio* extension was developed, which allows developers to manage and refactor *GreenPepper* acceptance tests from within the IDE. This supports the application of the *Executable Acceptance Test Driven Development* (EATDD) approach, in which all features of the system are specified by *Acceptance Tests*. So far, the extension only supports the refactoring of only a subset of the test specifications that the *GreenPepper* framework offers. With further development, these limitations could be eliminated.

6.3 Future Work

The following list mentions limitations of the current implementation as well as possible improvements that could be done in future work in order to eliminate these limitations.

- **Support for more GreenPepper tests**

  This thesis focused on implementing refactoring support for two different *GreenPepper* acceptance test types: The *RuleFor* test and the *Scenario* test. However, the *GreenPepper* framework offers a bigger set of test types. Future work could focus on extending the refactoring support by these test types.

- **Support for list notation**

  GreenPepper acceptance tests can not only be expressed in HTML tables, but also in bullet list form. Since the current implementation does only support HTML tables, it could be extended by also supporting the bullet list notation.

REFERENCES

[Ambler 2009a] Ambler, Scott W. ; *Examining the Agile Manifesto*, 2009, Online (Last cited: 18/02/2010), http://www.ambysoft.com/essays/agileManifesto.html

[ASE 2009] Agile Software Engineering Group, University of Calgary; *Executable Acceptance Test Driven Development*, 2009, Online (Last cited: 01/03/2010), http://ase.cpsc.ucalgary.ca/ase/index.php/EATDD/Home

[Astels 2003] Astels, David; *Test-driven development - A Practical Guide*, Prentice Hall, 2003, , ISBN: 0-13-101649-0

[Avery 2005] Avery, James; *What Is Visual Studio*, 2005, Online (Last cited: 10/03/2010), http://windowsdevcenter.com/pub/a/windows/2005/08/22/whatisVisualStudio.html?page=1

[Beck et al.] Beck, Kent; Andres, Cynthia; *Extreme Programming Explained, Embrace Change*, Addison-Wesley, 2004, Second Edition, ISBN: 0-321-27865-8

[Fowler et al. 2000] Fowler, Martin; Beck, Kent; Brant, John; Opdyke, William; Roberts, Don; *Refactoring: Improving the Design of Existing Code*, Addison Wesley, 2000, , ISBN: 0-201-48567-2

[GP Doc] Pyxis Technologies; *GreenPepper Documentation*, , Online (Last cited: 10/02/2010), http://www.greenpeppersoftware.com/confluence/display/GPWODOC/Documentation

[GP FAQ] Pyxis Technologies; *GreenPepper FAQ*, , Online (Last cited: 05/01/2010), http://www.greenpeppersoftware.com/confluence/display/GPW/FAQ

[GP Home] Pyxis Technologies; *GreenPepper Homepage*, , Online (Last cited: 06/01/2010), http://www.greenpeppersoftware.com/confluence/display/GPW/Home

[Gunnerson 2000] Gunnerson, Eric; *C#: Die neue Sprache für Microsofts .NET-Plattform*, Galileo Computing, 2000, 1st Edition, ISBN: 978-3898421072

[Koehler 2007] Koehler, Achim; *C/C++ Projektbegleiter: C/C++ Projekte planen, dokumentieren, bauen und testen*, DPunkt, 2007, 1. Edition, ISBN: 978-3-89864-470-9

[Kuehnel 2008] Kühnel, Andreas; *Visual C# 2008: Das umfassende Handbuch*, Galileo Computing, 2008, 4th Edition, ISBN: 978-3-8362-1172-7

[Manifesto] ; *Manifesto for Agile Software Development*, 2001, Online (Last cited: 18/02/2010), http://agilemanifesto.org/

[Maurer et al. 2005] Maurer, Frank; Read, Kris; Melnik, Grigori; *Student Experiences with Executable Acceptance Testing*, 2005, Proc. Agile 2005 Conference, IEEE Press, 2005

[Maurer et al. 2007] Maurer, Frank; Melnik, Grigori; *Multiple Perspectives on Executable Acceptance Test-Driven Development*, 2007, Proceedings of the 8th International Conference on Agile Processes in Software Engineering and eXtreme Programming (XP 2007), Como, Italy 2007

[Maurer et al. 2008] Maurer, Frank; Park, Shelly; *The Requirements Abstraction in User Stories and Executable Acceptance Tests*, 2008, Agile Conference 2008 (Research-in-Progress Workshop), Toronto

[Maurer et al. 2009] Maurer, Frank; Khandkar, Huq Shahedul; Park, Shelly: Ghanam, Yaser; *FitClipse: A Tool for Executable Acceptance Test Driven Development*, 2009, In Proc. of 10th International Conference on Agile Processes and eXtreme Programming (XP 2009), Demo Abstract, Pula, Italy, 2009

[Maurer et al. 2009a] Maurer, Frank; Park, Shelly; Ghanam, Yaser; Khandkar, Shahedul Huq;

FitClipse: A Tool for Executable Acceptance Test DrivenDevelopment, 2009, In Proc. of 10th International Conference on Agile Processes and eXtreme Programming (XP 2009)

[MSDN 2010a] Microsoft Corporation; *NET Framework Regular Expressions*, 2010, Online (Last cited: 09/03/2010), http://msdn.microsoft.com/en-us/library/hs600312%28VS.100%29.aspx

[MSDN 2010b] Microsoft Corporation; *Introducing the Visual Studio SDK*, 2010, Online (Last cited: 09/03/2010), http://msdn.microsoft.com/en-us/library/bb286983%28VS.100%29.aspx

[MSDN 2010d] Microsoft Corporation; *Automation and Extensibility Overview*, 2010, Online (Last cited: 07/03/2010), http://msdn.microsoft.com/en-us/library/aa290342%28VS.71%29.aspx

[MSDN 2010e] Microsoft Corporation; *How to: Get References to the DTE and DTE2 Objects* , 2010, Online (Last cited: 07/03/2010), http://msdn.microsoft.com/en-us/library/68shb4dw %28VS.80%29.aspx

[MSDN 2010f] Microsoft Corporation; *Visual Studio Command Table (.Vsct) Files*, 2010, Online (Last cited: 07/03/2010), http://msdn.microsoft.com/en-us/library/bb164699%28VS.100%29.aspx

[Nayyeri 2009b] Nayyeri, Keyvan ; *Visual Studio Add-In vs. Integration Package - Part 2*, 2009, Online (Last cited: 09/03/2010), http://nayyeri.net/visual-studio-addin-vs-integration-package-part-2

[Nayyeri 2009c] Nayyeri, Keyvan ; *Visual Studio Add-In vs. Integration Package - Part 3*, 2009, Online (Last cited: 09/03/2010), http://nayyeri.net/visual-studio-addin-vs-integration-package-part-3

[Nayyeri 2009d] Nayyeri, Keyvan ; *Visual Studio Add-In vs. Integration Package - Part 4*, 2009, Online (Last cited: 09/03/2010), http://nayyeri.net/visual-studio-addin-vs-integration-package-part-4

[Ordelt 2008] Ordelt, Heiko; *Refactoring of Acceptance Tests*, Hochschule Mannheim, 2008

[Power 2006] Power, Gus; *Values, Practices & Principles*, 2006, Online (Last cited: 24/02/2010), http://blog.energizedwork.com/2006/12/values-practices-principles.html

[Pyxis] Pyxis Technologies; *Pyxis Technologies Homepage*, , Online (Last cited: 11/02/2010), http://www.pyxis-tech.com/en/

[Pyxis Paper] Andre Brissette, Francois Beauregard; *Build the right software*, 2007, Online (Last cited: 04/02/2010), http://www.greenpeppersoftware.com/confluence/download/attachments/5/accurate_developement_wp_en.pdf

[Wells 2009] Wells, James Donovan; *Extreme Programming*, 2009, Online (Last cited: 24/02/2010), http://www.extremeprogramming.org/