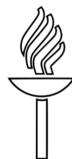
Nikolai Koudelia

Acceptance Test-Driven Development

Master's Thesis
in Information Technology
(Software engineering)
December 6, 2011



UNIVERSITY OF JYVÄSKYLÄ
DEPARTMENT OF MATHEMATICAL INFORMATION TECHNOLOGY

Jyväskylä

Author: Nikolai Koudelia
Contact information: nikolai.koudelia@gmail.com
Title: Acceptance Test-Driven Development
Työn nimi: Hyväksymistestivetoinen kehitys
Project: Master's Thesis in Information Technology (Software engineering)
Page count: 102
Abstract: Acceptance Test-Driven Development (ATDD) is meant to bring customers, engineers, testers and software developers together and help them understand each other. ATDD is not an automatic testing technique but rather a precise requirement management and software development convention, which helps to avoid misunderstandings between stakeholders and leads to production of program code satisfying only the real requirements making it more simple and clear. Conclusions about benefits and common problems emerging during adoption of ATDD are based on results of a software project which took place while this work was being written. They also concur very well with other case studies. A large framework of acceptance tests was built during the project. The framework turned out to be a good basement for production of high-quality program code, its maintenance and control of software complexity. Unfortunately, utilization of ATDD as a communication tool between software developers and other stakeholders failed completely.
Suomenkielinen tiivistelmä: Hyväksymistestivetoisen kehityksen (engl. ATDD) tarkoituksena on tuoda ohjelmistojen tilaajat, suunnittelijat, testaajat ja ohjelmoijat lähelle toisiaan ja antaa heille paremmat edellytykset toistensa ymmärtämiseen. ATDD ei ole automaattitestaustekniikka vaan pikemmin täsmällinen, osapuolten välille syntyvät väärinkäsitykset ennaltaehkäisevä vaatimusmäärittely- ja kehityskäytäntö, joka johdattaa vain todelliset tarpeet tyydyttävän ja sitä kautta selkeämmän ohjelmakoodin tuottamiseen. Työssä esitetyt päätelmät ATDD:n hyödyistä ja sen kääyttöönotossa ilmenevistä ongelmista perustuvat tämän työn ohessa suoritetun ohjelmistokehitysprojektin tuloksiin ja samalla yhtyvät hyvin paljon muualla tehtyihin tapaustutkimuksiin. Testivetoisesti kehitetyn projektin tuloksena syntyi laaja hyväksymistestikehys, joka antoi hyvät puitteet laadukkaan ohjelmistokoodin tuottamiseen, ylläpitoon ja monimutkaisuuden hallintaan. ATDD:n hyödyntäminen viestintävälineenä ohjelmistokehittäjien ja muiden osakkaiden välillä epäonnistui puolestaan täysin.
Keywords: ATDD TDD BDD AAT Acceptance Test-Driven Development FitNesse Fit
Avainsanat: testivetoinen hyväksymistestivetoinen

Contents

1 Introduction **1**

2 Background **4**

2.1 Waterfall model 5

2.2 V-Model 8

2.3 Common issues of plan-driven development 10

3 Why software fails **13**

3.1 Particular reasons 13

3.2 Why Big Software Projects Fail: The 12 Key Questions 15

4 Agile software development **20**

4.1 Agile Manifesto 20

4.2 Scrum 22

5 Introduction to ATDD **25**

5.1 ATDD process 25

5.2 Shared communication medium 26

5.3 Up-to-date documentation 27

5.4 Automatic testing and maintenance 28

5.5 Summary of ATDD goals 30

5.6 ATDD pitfalls 30

6 Test-Driven Development **33**

6.1 TDD is not about testing! 33

6.2 TDD work-flow 34

6.3 The goal of TDD 35

6.4 TDD pitfalls 35

6.5 Acceptance testing opposed to unit testing 37

7 Unit testing with NUnit **41**

7.1 Requirement elicitation 42

7.2 Example unit test 43

8 Behaviour-Driven Development **51**
8.1 The motivation behind BDD 51
8.2 Behaviour over implementation 52
8.3 Complex behaviour doesn't come alone 53

9 Acceptance testing with FitNesse **56**
9.1 ATDD vs GUI testing 56
9.2 Syntax of FitNesse tests 60
9.3 Specification scenario 61
9.4 A real example of FitNesse test 62
9.5 FitNesse tables 66
9.6 The Big Problem 69

10 The Missing Link **72**
10.1 Existing tools are just not good enough 72
10.2 Sketching a better solution 73

11 Available literature and case studies **79**
11.1 Common knowledge about AAT 80
11.2 Customers don't write tests 81
11.3 Other common issues 83
11.4 Benefits for development process and adoption costs 84
11.5 Test-driven requirement elicitation 86

12 Summary **88**

References **93**

1 Introduction

Software testing is obviously one of the most important components of the whole process of software development. Without proper testing none of the application's stakeholders can be sure whether it works as expected or doesn't. Nevertheless, a situation when deadlines are approaching but the project is still in the middle of construction phase is very common. In such cases the construction phase usually continues for as long as possible, sacrificing the testing phase at the same time. As a result, customers get their software untested and full of bugs.

This work discusses an alternative technique the main purpose of which is to split testing across all the life cycle of software, starting from elicitation of requirements and continuing throughout construction and maintenance phases. Such a thorough transformation of the convenient post-constructional testing activity turns it into something else but testing. Such a technique stops from being a testing activity and becomes test-*driven* development. Test-driven development doesn't fully remove the need of applying post-constructional testing but greatly simplifies such testing by moving the most laborious parts of it away from human testers to be automatically executed by computer as often as needed. Properly conducted test-driven development makes a "bug" become the exception, not the rule.

Chapter 2 discusses different types of software complexity and lays the ground for clear understanding of the needs behind development techniques discussed in this work. It also implies the obstacles on the way of adoption of these techniques. Finally, the chapter introduces two plan-driven development models — Waterfall and V-models — and discusses the most common drawbacks of these techniques.

Chapter 3 reviews the most common reasons for failures of large-scale development projects and discusses the suggestions how to overcome these issues. The chapter serves as a preface for introduction of Agile software development.

Chapter 4 introduces Agile software development and Agile Manifesto which offer an alternative way of developing software. The most popular agile development model, Scrum, get also introduced in the chapter. It also draws a distinction between true software development models like Waterfall, V-Model and Scrum and development *practices* which are the main subject of this work – ATDD and TDD.

Chapters 5 and 6 introduce ATDD and TDD, respectively. They explain the different goals of these techniques, their pitfalls and development processes. The latter

chapter explains in detail two essential ideas causing a lot of confusion to people who are not familiar with fundamental objectives behind them.

The first idea is about test-driven development not being a testing technique; automatic tests only become "tests" after having implemented all the functionality being test-*driven*. Before that there is nothing to actually *test* and those artifacts rather serve as executable *specifications* but not tests.

The second idea is about the fundamental difference between TDD and ATDD: the former is solely practiced by programmers to keep program code clean and to help them discover optimal solutions for code-centric problems. The latter in turn is meant to be practiced by both programmers and domain experts in order to establish shared understanding about the problem being solved. Additionally, there are some other factors driving the techniques apart which are explained in the ending part of Chapter 6.

The problem with these two fundamental concepts is harder than it appears to be because the actual development process is the same for both techniques and they still have many thing in common. Consequently, it's hard to tell in the beginning, which kinds of features should be test-driven with which one of the techniques, how much functionality should be implemented in one go and whether applying a technique is rational in the first place. Comprehending the concepts discussed above helps to answer these questions which are even partially irrelevant.

Chapters 7 and 9 represent real-life examples of unit (TDD) and acceptance tests (ATDD) and try to further clarify the difference between the techniques. Chapter 8 serves as a link between these two chapters and introduces Behaviour-Driven Development (BDD), a technique which was born because of the abstract nature of unit tests. BDD was introduced in 2006 by Dan North who wanted to shift the balance from code-centric TDD toward behaviour-oriented BDD. However, BDD doesn't suite all kinds of software – it emphasizes simplicity of test representation and abstracts (hides inside program code) irrelevant test data away making it inaccessible for domain experts. Such a technique doesn't work for data-rich applications like financial software normally having too much significant data which can't be hidden.

Chapter 9 proceeds with an example of tabular acceptance test implemented with an ATDD tool FitNesse. Including high amounts of test data has its price though: the tests become much more complex and harder to follow, exactly what BDD was trying to avoid. Because hiding the data is not an acceptable solution, it has to be managed. FitNesse has a powerful mechanism for test management. It makes it easy to organize the tests into a hierarchical order, to search inside test data and to refactor tests (mainly rename them). The problem arises when editing large

amounts of test data with the built-in text editor which is not suitable for editing tabular data.

Chapter 10 further reviews the problems the author of this text stumbled upon during the project. In order to make the problem more vivid and clear, the author describes an imaginary ATDD tool having the most important features he thinks a proper ATDD tool should have.

Chapter 11 contains a review of three articles written by Børge Haugset and Geir K. Hanssen. The articles describe case studies conducted by the authors as well as a literature review and results of case studies performed by other researchers.

The chapter starts by mentioning about poor availability of high-quality academic research material concerning ATDD which is also confirmed by Haugset and Hanssen. Next, the main outcomes of the studies are discussed. There are two of them: first, the case studies confirm, that FitNesse *can* be successfully used in software development; it makes development much more secure and easier to refactor which, in turn, leads to better quality. The second main outcome is about customers not being generally interested in writing executable acceptance tests.

The chapter proceeds with reviewing the articles by presenting the description of AAT (Automated Acceptance Testing) principles and deepening into the most common issues and benefits of AAT adoption.

Although the articles were discovered by the author of this work at the final stage of this work's writing, the conclusions made throughout this work agree very well with the results from case studies discussed in the articles.

The last chapter summarizes each chapter of this work in a similar way with this chapter, but in a slightly more accurate way. It also includes the author's personal sentiments about this work and the software project taken place while writing this thesis.

2 Background

Frederick Brooks wrote his famous essay "No Silver Bullet: Essence and Accidents of Software Engineering" [1] in 1986. The basic idea of the essay is about complexity being an essential part of software and thereby building software will never be an easy task:

> "The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions. This essence is abstract in that such a conceptual construct is the same under many different representations. It is nonetheless highly precise and richly detailed.
>
> I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation. We still make syntax errors, to be sure; but they are fuzz compared with the conceptual errors in most systems. If this is true, building software will always be hard.
>
> There is inherently no silver bullet."

Brooks divides software complexity into two subcategories: essential and accidental complexity.

Accidental complexity involves issues which can be avoided by using appropriate tools or development processes. Deployment of high-level languages, which reduce accidental complexity, is probably the biggest breakthrough in software industry. It's hardly possible and obviously not rational to implement an average GUI application which utilizes database completely in assembly language. High-level languages let developers concentrate on the actual problem instead of manipulating processor registers. Another ubiquitous concept attempting to make program code more human-friendly is object-oriented languages. As well as high-level languages, object-oriented programming is meant to facilitate the task of expressing business domain problems as machine code by reducing accidental complexity. Leaping ahead, accidental complexity is going to become the major problem during this work, too.

While accidental complexity can be eliminated by proper tools, essential complexity is inevitable. Brooks does not provide a clear definition of what essential complexity actually is, but the article describes it as a natural property of software in general and as an inevitable consequence of complex software requirements. The complexity of software cannot be eliminated without sacrificing requirements which is normally not acceptable. Brooks puts it as follows:

> ”The complexity of software is an essential property, not an accidental one. Hence, descriptions of a software entity that abstract away its complexity often abstract away its essence.”

The citation above means there is no easy way (i.e. silver bullet) to make computer implement a solution for any given problem automatically and that complexity is going to remain an imperative property of software in general. Brooks mentions automatic and graphical programming as only suitable for specific tasks, not general ones. Things appear to have remained the same after over twenty years since his writing.

Not being able to automate software engineering, however, doesn't mean essential complexity can't be made easier to manage. The purpose of this work is to introduce Acceptance Test-Driven Development (ATDD), a test-driven software development technique the basic meaning of which is to help managing essential complexity. ATDD provides some alternative ways of solving the ancient challenges of software engineering — communication, documentation and confirmation of the system meeting its requirements. Nevertheless, the importance of these activities has been recognized long before the concept of ATDD has been introduced, at least over forty years ago, in the beginning of Waterfall model's era. ATDD and other related techniques are going to be examined throughout this work but first, Waterfall model (and why it doesn't work too well) will be discussed.

2.1 Waterfall model

Dr. Winston W. Royce, who was a director at Lockheed Software Technology Center in Austin, Texas, was among the first computer scientists to describe a technique similar to what now is known as Waterfall model in his article ”Managing The Development Of Large Software Systems” in 1970 [2]. Although Royce was among the pioneers of plan-driven development, he managed to foresee the major problems and at the same time, as Royce puts it, the necessary preconditions of a successful plan-driven software development process.

Royce starts his article cited above by pointing out the two absolutely necessary steps of a software development project regardless of size and complexity of a software — the analysis and coding. Next, Royce introduces the other steps (Figure 2.1) he considers essential in order to succeed in software development with respect to time and budget limits.

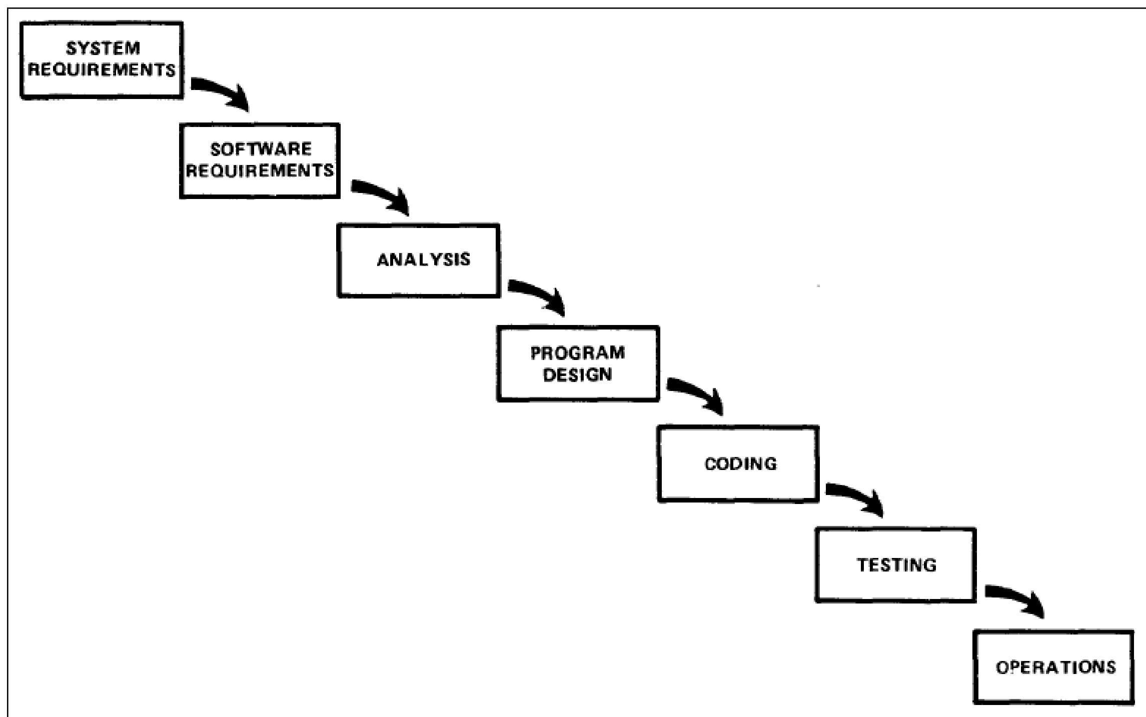


Figure 2.1: Waterfall model

Leaping ahead, the model described by Royce in not a genuine "waterfall" model, because the final version of the model allows movement in both directions, down *and* up. In fact, Royce never even called the model by that famous name in the article. Nevertheless, the author is going to follow the common malpractice and use the term "waterfall" just to avoid calling it "the model".

Waterfall model includes two major steps additionally to general analysis and coding — the software requirements and testing. There are also *system requirements*, *program design* and *operations* steps (Figure 2.1) but these are less important in respect of the whole process and may be combined into more generic entities as in Figure 2.2.

A one-way waterfall model would be a perfect process if it was possible to design a perfect solution and then perfectly implement it. The phase of testing wouldn't be necessary at all as a perfect implementation would guarantee satisfaction of all requirements. A common scenario however involves the testing phase which reveals

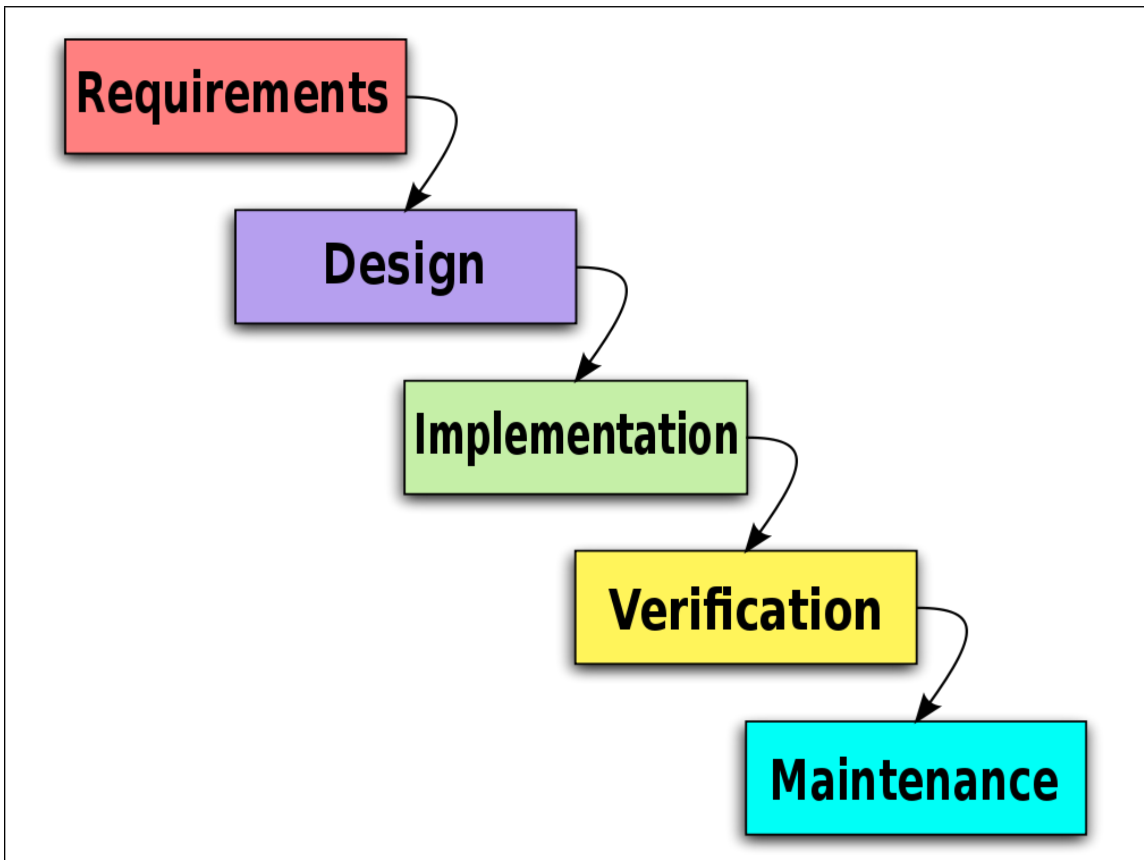


Figure 2.2: Waterfall model

flaws in some of the earlier phases which have to be re-done as illustrated in Figure 2.3.

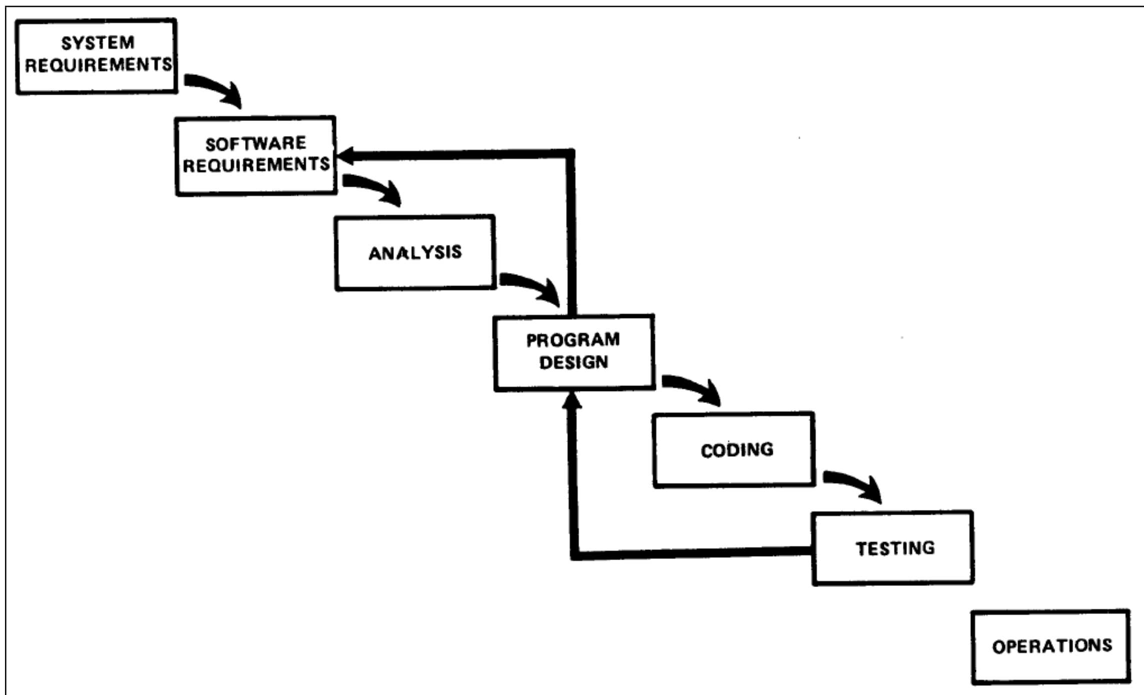


Figure 2.3: Iterative waterfall model

The problem with such approach is that the testing phase may be postponed until the project deadline if the progress gets stuck iterating at implementation and design phases. Now not only there is less time available for testing, there also won't be enough time to fix all the bugs found during the testing phase.

According to The Standish Group's report "CHAOS summary 2009" [6], only 32% of all software projects success, 44% are challenged (i.e. late, over budget, and/or with less than the required features and functions) and 24% are failed meaning they are cancelled prior to completion or delivered and never used.

2.2 V-Model

The previous section described Waterfall model where each subsequent step is only connected to a previous one. This section is going to briefly discuss another plan-driven software development technique which may be considered an extension of the Waterfall model – the V-model. This section also tries to describe in which way test-driven techniques being covered in this text relate to V-model.

V-model differs from Waterfall model mainly in two ways. First, it divides the

testing phase into smaller particles: unit testing, system verification and system validation [4]. Second, V-model connects pre-construction phases like requirement elicitation and design to the corresponding post-construction testing phases. Figure 2.4 shows an illustration of V-model from the system engineering guide issued by the U.S. department of transportation (Systems Engineering for Intelligent Transportation Systems) [4][p.11].

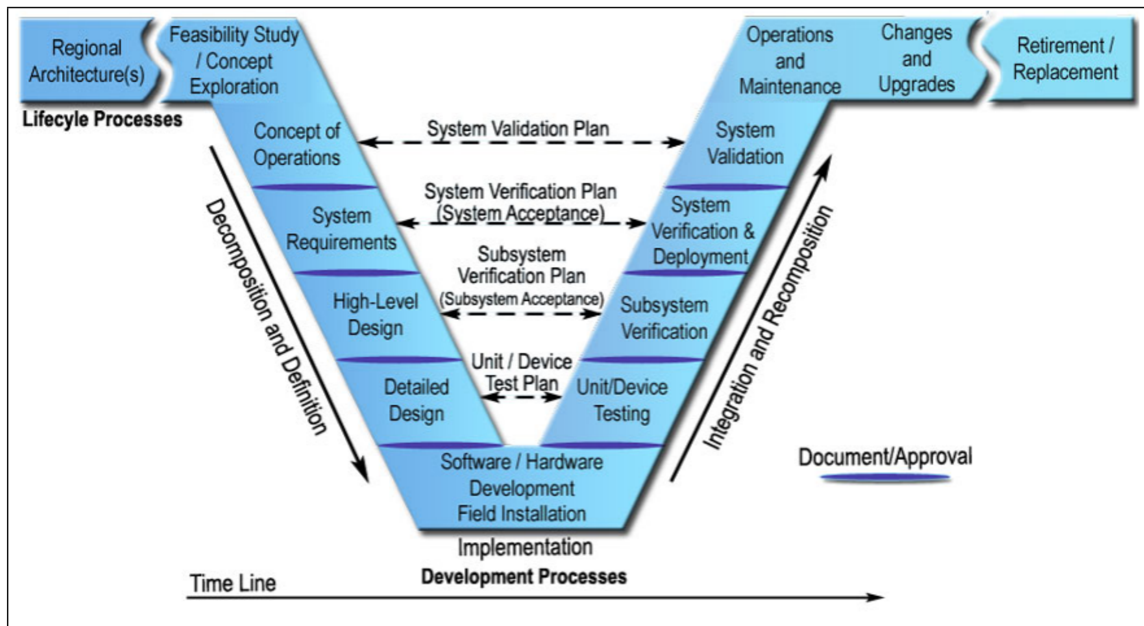


Figure 2.4: V-Model

According to V-model, each pre-construction phase involves a detailed planning of a corresponding testing phase. For example, the "Detailed Design" not only includes the low-level design of program code units but also the planning of how each code unit is going to be tested. Each phase on the right wing of the "Vee" is hence connected and planned with respect to a corresponding phase on the left wing.

The different testing phases described in the ITS guide are meant to test software systems at different levels. The first post-construction testing phase is Unit/Device Testing. Its goal is to assure that each separate software module was built according to specifications produced during the Detailed Design and doesn't contain any "bugs", i.e. accidental implementation errors.

The two following testing phases (Subsystem Verification and System Verification & Deployment) test for distinct modules to work in assemblies and for the whole system's conformity to its requirements.

The last testing phase — System Validation — is meant to ensure the system

actually satisfies business needs. ITS guide [4][p.60] provides a good explanation of how *verification* and *validation* differ from each other:

> In systems engineering, we draw a distinction between verification and validation. Verification confirms that a product meets its specified requirements. Validation confirms that the product fulfills its intended use. In other words, verification ensures that you "built the product right", whereas validation ensures that you "built the right product".

Both Waterfall and V-model are complete software development models covering the "big picture". They guide through the general concepts of software development and cover the main activities to be done during a project, all the way from requirements to maintenance.

Test-driven techniques discussed throughout this work operate on a lower level instead. They provide tools to be applied in certain development phases of a bigger scale. In case of V-model, ATDD operates on the three upper levels shown in Figure 2.4: Concept of Operations and System Validation, System Requirements and System Verification & Deployment, High-Level Design and Subsystem Verification.

TDD (Test-Driven Development), in turn, covers the bottom levels: Detailed Design, Unit/Device Testing and Implementation.

The following section is going to briefly introduce the two main subjects of this work: insufficient testing and lack of up-to-date documentation. These are the most common problems of plan-driven development and, respectively, the high-level goals of all test-driven techniques.

2.3 Common issues of plan-driven development

Returning to software documentation, Royce states [2][p.332] right from the start:

> "The first rule of managing software development is ruthless enforcement of documentation requirements."

He further adds that when a software documentation gets into bad shape, the first thing he advices to do is replacing the project management and stopping all activities not related to documentation. Royce lists the main reasons for the importance of documentation: first, documentation provides the communication medium for team members. It also serves as a progress indicator providing some real evidence of what features have been implemented instead of "90-percent ready" promises

month after month. Second, in the beginning of a software project, the documentation *is* the specification and *is* the design. If the documentation is bad, then the design is bad; the three words — documentation, specification and design — all mean the same thing. Finally, documentation is an invaluable foundation for testing.

Figure 2.5 presents the variety of documents suggested by Royce in his article [2][p.333]. There are six main types of documents some of which need to be written several times during project lifetime.

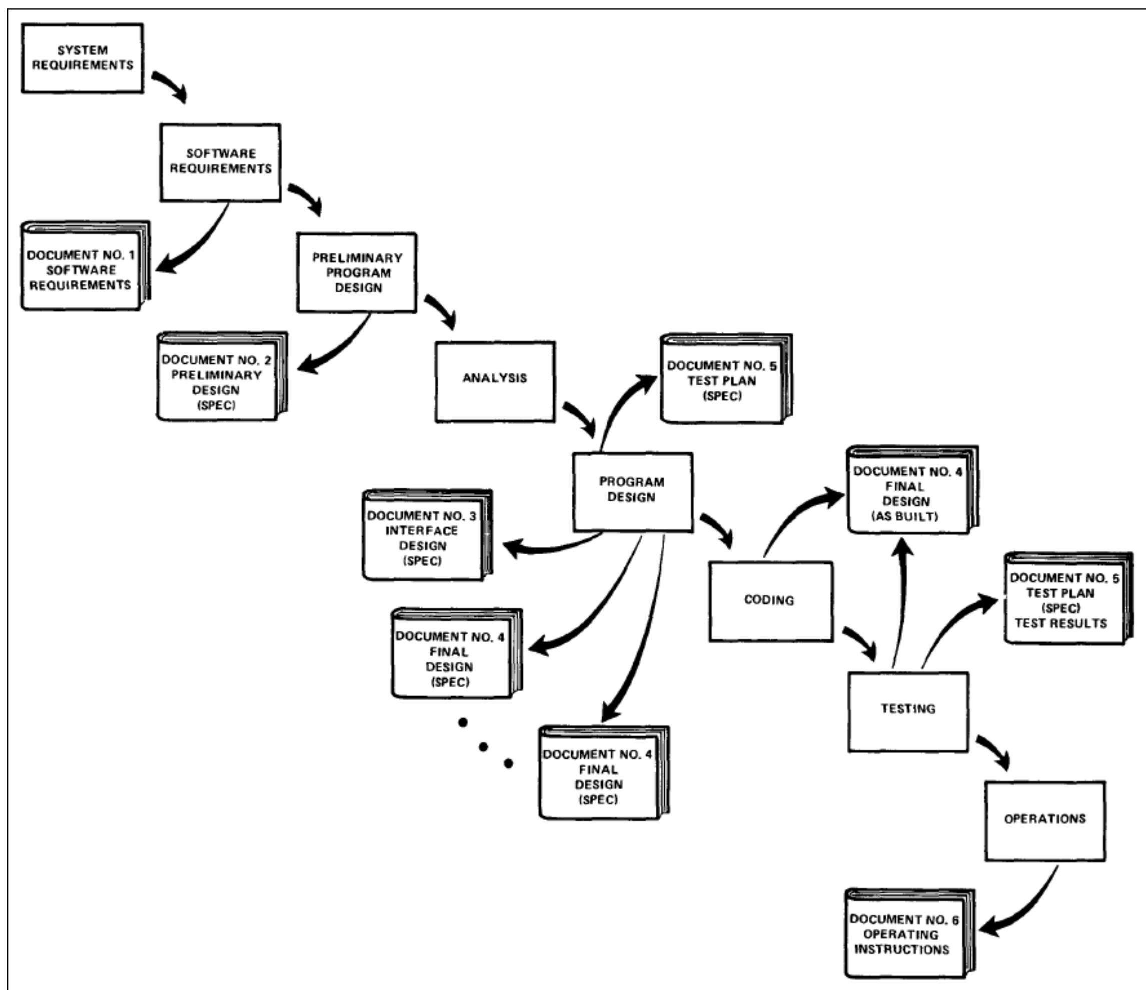


Figure 2.5: Documentation documents

Royce highlights the importance of up-to-date documentation for a reason. Despite of all the benefits of a good documentation, it's very easy to give up on. The problem is that maintaining such a rainbow of documents demands strict discipline which is probably not the way most programmers would like to do their job. Additionally, it's not even possible to continuously track ever-changing program code

and reflect all those changes to static documentation. Static documentation is "invisible" with respect to changing program code. Changes to code don't affect specifications or any other documents directly, but instead, a human interaction is always needed to evaluate the changes and inspect the documentation for inconsistencies. As a result of humans' inability and unwillingness to manually maintain the conformity between program code and its documentation, change by change, documentation becomes even harder to synchronize with program code and more likely to be archived (i.e. abandoned).

Another big issue with plan-driven development is it's practically impossible to stick to original plan. The model doesn't take into account the fact that complex designs are too complex to be fully designed before implementation. Consequently, the implementation phase reveals problems which were not payed enough attention in the design phase requiring to re-evaluate the design. As a result both design and implementation take more time than it was initially planned. The worst part is that the deadline doesn't normally get adjusted according to reality but remains the same. Hence, the last phase of the Waterfall model — testing — gets sacrificed by giving up the time originally dedicated to it to the earlier phases. In other words, customers get their software untested and full of bugs.

These two problems — lack of testing and up-to-date documentation — are the main improvement targets of ATDD, a technique which is going to be evaluated in detail throughout this work, but first, the motivation to switch away from plan-driven development models should be better explained. The following chapter reflects the most common reasons behind failures of large-scale software projects.

3 Why software fails

This chapter is going to discuss the most common reasons behind software failures. It slightly overlaps with the previous chapter while describing the traditional waterfall development process. The goal is to continue from general description of Waterfall model to particular failure reasons. This chapter also serves as a basement for reflections about possible improvement measures and the last section is a preface for those measures, namely agile software development.

3.1 Particular reasons

The process of software development consists of the following main activities: requirement engineering, design, implementation, testing and maintenance. Traditionally these tasks are executed in that order [5]. Probably the most widespread software development model, the waterfall model, applies the tasks stated above in that particular sequence too. Waterfall model is iterative, i.e. development cycles validate each other so that conflicting phases may be executed in reverse order. For example, a failing test forces the process to return to the "implementation" phase which in turn may require a review of the "design" phase. This kind of process seems natural and logically should lead to satisfactory results. The real world is not that simple though. Requirements are not always clear, testing does not always reveal all the problems or lack thereof and iterating is not any more possible at some stage of a project.

Despite of passed decades of evolution of software development, low quality is still an issue [6]. New languages, better compilers and object-oriented programming have improved software development tools but it doesn't seem to be enough. Bug tracking systems are still a common thing because big projects tend to have thousands of defects and tracking them is impossible without proper tools.

The name of this chapter is borrowed from the article written by Robert N. Charette [7]. The article presents several dozens cases of failed software projects costing astronomical money figures. The huge degree of "chaos" plaguing the IT industry brought up by Charette's article is also confirmed by the Standish Group's CHAOS reports. The 1995 CHAOS report [3] presents a number of $81 billions of dollars lost by US companies due to 31.1% of all projects being cancelled and 52.7%

costing almost twice as much as it was originally estimated. In 2002 the percentage of cancelled projects dropped to 15% but gradually increased during the following years: 18% in 2004, 19% in 2006 and 24% in 2008 [6]. Only a third of all projects on average was reported to be successful during the last decade and the remaining two thirds being either cancelled or finished with significant overrun in budget and/or time.

So, why software projects tend to fail? Charette provides the following reasons for failures of software projects:

- Unrealistic or unarticulated project goals
- Inaccurate estimates of needed resources
- Badly defined system requirements
- Poor reporting of the project's status
- Unmanaged risks
- Poor communication among customers, developers, and users
- Use of immature technology
- Inability to handle the project's complexity
- Sloppy development practices
- Poor project management
- Stakeholder politics
- Commercial pressure

Some of the reasons listed above are self-explanatory while some of them are further explained in the article and supported by numerous real-life examples [7]. While each of the statements listed above seems reasonable, they look like obvious answers to the question "What might cause failures of software projects". The article cites several case studies which identify failure reasons by surveys, in other words by asking people why the projects they worked on failed. The people behind projects' failures are probably not a very reliable source of information as they'd likely try to blame someone else but themselves. For instance, the list doesn't include such fundamental failure reasons as lack of skill, motivation and commitment. Even "Inability to handle the project's complexity" looks like an accusation against management of letting the project become too complex.

3.2 Why Big Software Projects Fail: The 12 Key Questions

Watts S. Humphrey, who was often called the "Father of software quality" (mentioned in the Wikipedia article about Watts Humphrey [8]), had a different approach to the subject in his article "Why Big Software Projects Fail: The 12 Key Questions" [9]. The article consists of a series of questions considering common software development challenges and reflections on measures required to overcome those challenges.

The article [9] starts by pointing out how high the average failure level of software projects is. Humphrey further cites The Standish Group's data indicating that the level of success drops quickly as the project size grows — half of the smallest projects succeed while all the largest projects fail (Figure 3.1).

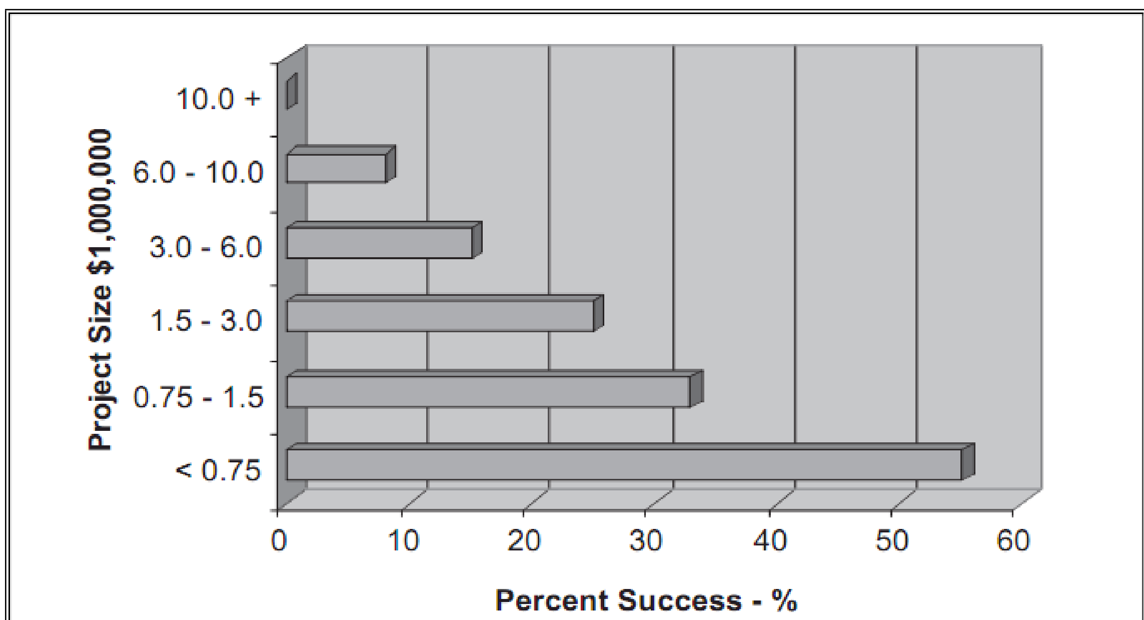


Figure 3.1: Success rate by project size

Short summaries of each key question from the article [9] follow next. All the ideas in the following review are taken from the article and will not be explicitly cited.

Question 1: Are All Large Software Projects Unmanageable?

Humphrey answers that question by referencing two large successful projects: Command Center Processing and Display System Replacement (CCPDS-R) project and the operating system (OS)/360 project. The former had about 100 developers involved and the latter had about 3000 software professionals.

Both project share many common principles; both were thoroughly planned and adopted evolutionary development strategy with multiple releases and phased specifications. Humphrey notes that big projects must be build in small, iterative steps. This agrees with the sixth law of CHAOS: ″Law Of The Edible Elephant — the only way to eat an elephant is one bite at a time″ [6]. This is also one of the twelve principles of Agile Manifesto [10] — ″Working software is delivered frequently (weeks rather than months)″ and is the basic idea behind ATDD.

Question 2: Why Are Large Software Projects Hard to Manage?

Humphrey reviews the history of mankind with respect to management form of big projects. Most of them have been led by authorities, especially construction and manufacturing projects. There have been exceptions like cathedrals which were built mostly by volunteer artisans who managed themselves under the guidance of a master builder. Humphrey compares such projects to open-source software projects which are developed by volunteer programmers too. The vast majority of in-house projects are however still autocratic.

Although autocratic and hierarchical management style proved to be effective for management of physical work like building and factory production, it doesn't work that well for management of large-scale software projects.

Question 3: Why Is Autocratic Management Ineffective for Software?

The major problem with authoritative management system concerns management visibility. An army officer or a manager of a factory can instantly see what the subordinates are doing, what is the progress state and whether some individuals a being unproductive. Software is different in a way that none of these things reveal by merely watching.

With software the progress may be estimated only by asking developers or by careful examination of what they have produced.

Question 4: Why Is Management Visibility a Problem for Software?

In order to manage any kind of large-scale projects, the management must know what is the current phase of the project, how rapidly the work is being done and what is the quality of the products being produced. Modern software projects often lack these crucial pieces of information.

Without such knowledge small problems add up over time without being noticed. When they are finally noticed it's already too late to start fixing each of them and the project is in serious trouble.

Question 5: Why Can't Managers Just Ask the Developers?

A logical conclusion is to ask developers of software what is the progress of their work. The problem however is that with current software practices developers don't

know where they stand any more than the managers do.

By ”current software practices” Humphrey means that programmers do not have personal plans, they do not measure their work and they do not track their progress. He further states that unless individual developers plan and track their work, that work will be unpredictable and as a consequence their projects will be uncontrollable and unmanageable.

Question 6: Why Do Planned Projects Fail?

The common problem with planned projects is that planning precision it too low. Projects only have major milestones such as specifications complete, code complete and the like. With such broad goals it’s very hard to do precise estimations of completion dates and the three main development phases (design, implementation and testing) overlap and in practice they all co-exist throughout whole project regardless of the plans.

Humphrey provides a case example of a large project he was asked to review. He was told that the code implementation phase has already been finished on schedule. Nevertheless, Humphrey found out that very little code had actually been released to test and the developers did not know how much code they had written and what remained to be done. The project took then ten more month before all the coding was actually completed.

The case confirmed that without objective data developers have no way to know precisely where they stand. Being afraid to bear bad news they rather tell the most optimistic story they can which is usually far from reality.

Question 7: Why Not Just Insist on Detailed Plans?

It’s obviously reasonable to have a good detailed plan, but as Humphrey states, ”Whose plans are they?” With hardware, construction and other processes producing physical products the detailed plans are composed prior the production phase. With high-technology work, particularly with software, this approach has become progressively less effective. The principal reason is that the managers don’t know enough about the work to make detailed plans and the plans are hence very generalized. The current system is therefore the modern equivalent of the cathedral-building system where the developers act like artisans.

Question 8: Why Not Tell the Developers to Plan Their Work?

If the developers know best how to make their own plans, then they should obviously do so. There are however some problems with such planning.

First, most developers do not want to make plans; they would rather write programs. They see planning as a management responsibility. Second, most developers don’t know how to make plans. Finally, making accurate, complete and detailed

plans means that the developers must be empowered to define their own processes, methods, and schedules.

Letting developers make their own plans would transfer a lot of responsibilities to them. The managers would not be willing to allow it unless they had evidence that the developers can produce acceptable results.

Question 9: How Can We Get Developers to Make Good Plans?

Humphrey brings out two essential questions: ”How can we get the software developers and their teams to properly make and faithfully follow detailed plans, and how can we convince management to trust the developers to plan, track and manage their own work?”

The three things to be done to get the developers make and follow their personal plans are: the developers must be provided with the skills to make accurate plans, they should be convinced to make these plans and support and guide them while they do it.

Question 10: How Can Management Trust Developers to Make Plans?

The biggest risk of all is: How can the developers be trusted to produce and follow their own plans which should meet the management's goals?

This is the main problem of autocratic management methods: trust. If programmers are trusted to manage themselves, they will do extraordinary work. However, trust cannot be blind. The management must ensure that programmers know how to manage their own work and monitor they do it properly.

Humphrey boils the trust question down: ”If you do not trust your people, you will not get their whole-hearted effort and you will not capitalize on the enormous creative potential of cohesive and motivated teamwork.”

Question 11: What Are the Risks of Changing?

There are two risks associated with changing a software development process. First, it costs time and money to train staff to adopt the new process. Second, the new process may turn out to be ineffective. There is no time for pilots during large projects so the development practices must be chosen in the beginning and used throughout the project. The efficiency analysis may be done only after the project is finished.

Although there are always some risks associated with any change, there is a cost for doing nothing too. Since most large software projects fail anyway, the biggest risk in *not changing*.

Question 12: What Has Been the Experience So Far?

Humphrey answers the last question by introducing the Team Software Process (TSP) which is a software development process following concepts described in the

twelve questions presented in this chapter [11]. TSP was initially developed by Humphrey and later by the Software Engineering Institute (SEI).

Humphrey writes: "With the TSP, if you properly train and support your development people and if you follow the SEI's TSP introduction strategy, your teams will be motivated to do the job properly." He further mentions that TSP has proved to be effective for teams of up to about 100 members some of which included multiple hardware, systems and and software professionals. TSP also has worked for distributed teams from multiple geographic locations and organizations.

The first eleven questions in Watts Humphrey's article [9] serve as a preface to the last "question" which is an introduction to the TSP, his own development process. Nonetheless, in the preface part Humphrey addresses and analyzes the common issues of large software projects leaning on all his experience and results of case studies.

To sum up, the main issues of large software projects presented in the Humphrey's article include bad management visibility and inability of software designers to provide and follow precise work estimates. These two problems are consequences of different factors such as the autocratic management system, the nature of software which is essentially complex, lack of trust and commitment and others. However, the most essential failure factor according to Humphrey is the inability to follow a precisely planned development process.

Humphrey seems to be somewhat in conflict with the Agile Manifesto and agile software development which are going to be introduced next. Agile techniques emphasize human interactions over processes and responding to change over following a plan. Even though Agile Manifesto tells to prefer human interactions over process, that doesn't mean there is no processes to follow in agile development. In fact, test-first agile techniques do follow strict yet simple test-code-review (or red/green/refactor) process.

4 Agile software development

This chapter proceeds with reflections about common reasons for project failures by introducing agile software development and discussing in which (different) ways things are suggested to be done in order to succeed in software development.

4.1 Agile Manifesto

The term "Agile software development" was introduced in 2001 by the authors (Kent Beck, Ward Cunningham, Robert C. Martin and others) of the Agile Manifesto [10]. The manifesto follows:

> "We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> - **Individuals and interactions** over processes and tools
> - **Working software** over comprehensive documentation
> - **Customer collaboration** over contract negotiation
> - **Responding to change** over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more."

There are many agile software development techniques existing. Probably the best known agile techniques are Extreme Programming [12] and Scrum ([13], [14]). Different agile techniques differ from each other by their goals and ways of achieving them. All of them however share the values represented in Agile Manifesto.

One of the basic ideas behind all agile techniques is that in most cases a complex design cannot be fully implemented in distinct phases suggested by Waterfall model and other plan-driven techniques. Agile techniques take into account the imperfect human nature and anticipate right from the beginning such a common things as incomplete requirements, buggy code and invalid testing. Instead of applying these activities as big separate chunks of software development process, agile techniques combine all of them – requirement elicitation, design, implementation and testing – in small pieces gradually building up the software. Agile techniques split testing

across all the development process instead of only applying it as a final verification activity. As the name suggests, ATDD is a test-driven practice meaning that each development cycle starts with implementation of an acceptance test and not of a feature itself [15][p.10] (Figure 4.1). Although test-driven (or test-first) development is the most common way of programming in agile methodology, the tests don't necessary have to always precede the construction — the essential concept is combining both programming and testing into a uniform process.

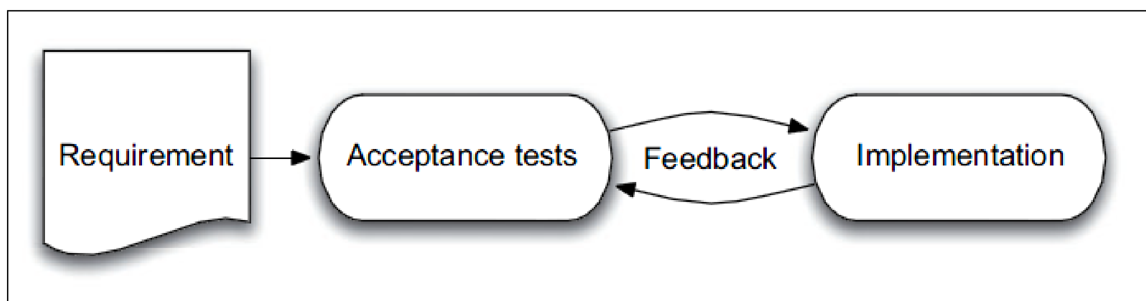


Figure 4.1: ATDD cycle

Including testing in development process not only aims at reducing the amount of testing to be done at the end, it also helps to gather and maintain software documentation.

The leading question is, are those ″tests″ actually tests any more?

The question is much trickier than it appears to be. The word ″test″ has a very precise, unambiguous meaning. However, depending on timing, these so-called tests get also utilized as specifications, not tests. According to Dan North [23], the ″father″ of Behaviour Driven Development (BDD), they shouldn't be called tests at all but rather ″executable specifications″.

Dan North introduced BDD in his article ″Introducing BDD″ in 2006 [23]. Both practices, ATDD and BDD are about the same thing: driving software development with executable specifications. North felt that the word ″test″ caused too much misunderstanding for people learning the practice as the word ″test″ strongly associates with the traditional post-constructional software testing. Hence North decided to remove the word ″test″ and to use ″behaviour″ and ″specifications″ instead, because by the time acceptance/unit tests get written there is actually nothing to test yet. This distinction is important understand in order to realize which ″tests″ should serve which kinds of artifacts and when to write or to skip writing tests. This problem is one of the most important topics of this work and will be further discussed throughout later chapters.

Nevertheless, this text got its name (Acceptance Test-Drive Development) in advance and it also took many essential ideas from Lasse Koskela's book ”Test Driven, Practical TDD and Acceptance TDD for Java Developers” [15]. It may therefore be appropriate to stick to ATDD as the top-level term but to use ”specifications” instead of ”tests” whenever such a swap makes sense.

Another core part of the concept of agile development is persistent involvement of domain experts in the process of software development. This means that domain experts don't disappear after they have done their job specifying the system, which commonly happens with waterfall projects. In agile development requirement providers are supposed to be available during all the time a software remains under construction. Moreover, they play the key role because every single increment involves requirement engineering including elicitation, validation and verification of whether the software meets its requirements (i.e. testing) after an increment has been implemented. As one can notice, agile development is all about communication – the incremental nature of agile development process ties team members together allowing each stakeholder group (meaning here domain experts, developers, testers and such) to affect the development process early enough when a correction cost is yet low.

4.2 Scrum

Scrum ([13], [14]) is probably the most wide-spread agile framework for project management. Scrum is not a development model or process, it doesn't tell exactly what and how to do in any given situation. Instead, scrum gives a team the power to choose the best possible way of working.

Scrum doesn't equate to total anarchy though. There is a handful of rules which form a skeleton providing necessary guidelines for keeping and improving team's productivity.

Scrum divides all the work to be done into small, concrete chunks which are meant to be fully completed during certain periods of time – *sprints*. The team decides how much work to promise to finish in a sprint and how long the sprint should be (usually 1-4 weeks). The goal is to choose only so much work in the beginning of a sprint that all of it gets ready by the end of the sprint.

The work is being pulled from *product backlog* (the main artifact of Scrum) onto sprint backlog. Product backlog is a list of all available items (or features, stories – anything the customer wants to call them) to be implemented by the team. Sprint backlog contains only those items selected for the current sprint. The main attributes

of each item are priority and relative complexity levels. The most important items are processed first. The complexity levels are determined by the team while the priority is set by the *product owner*.

Scrum has three roles: Product Owner, Team and *Scrum Master*.

Product owner is the domain expert who is responsible for providing the needed information about the product (software being developed) and for selecting which features are the most important and should be implemented first.

Team is a small, cross-functional, self-organizing group of people who work together to reach a common goal of finishing all the items on the sprint backlog. The term ”scrum” comes from rugby and, just like in rugby, the team ”tries to go the distance as a unit, passing the ball back and forth” [16].

Scrum Master supports the team by ensuring that external factors don't disturb the team and that the rules of Scrum don't get violated.

According to Google trends, Scrum is clearly the most popular agile software development method which suggests it can be extremely productive. However, like any method, Scrum can be misapplied. James Shore wrote in his blog article ”The Decline and Fall of Agile” [17] about failures with Scrum which are very common. Many teams adopt the main process described above at least to some extent, but that's not enough. They leave out the detailed planning phase of Waterfall model but don't replace it with anything to assure design evolution. They leave out conventional product documentation but don't replace it with high-bandwidth communication and automatic acceptance testing. They simply plan and re-plan one failed sprint after another. Shore compares the phenomena to unhealthy eating:

> ”These pseudo-Agile teams are having dessert every night and skipping their vegetables. By leaving out all the other stuff – the stuff that's really Agile – they're setting themselves up for rotten teeth, an over-sized waistline and ultimate failure. They feel good now, but it won't last.”

Shore stresses the importance of true Agility which involves such core attributes as agile engineering practices, high-bandwidth communication and a strong customer voice. This work concentrates on two techniques implied by Shore – TDD and ATDD [15][p.324] – to boost communication and to secure design evolution and code quality.

The software development techniques described in this work are not actually full-blooded agile *development models*. A true development model (like V-model, Waterfall or XP) covers development process at a high abstraction level: it defines

the management practices of a project, member roles and their responsibilities, documentation items and other rules and conventions. It may be considered team's constitution.

The main techniques discussed in this text (TDD, BDD, ATDD) are in turn the actual tools in hands of agile teams' members. They get utilized at the point when some program code is going to be produced or modified or the system under test is going to be validated for meeting its requirements. Although these techniques are not agile development *models*, they are designed to be practiced in conformity with the values declared in the agile manifesto.

The following chapter is going to introduce ATDD, the main subject of this work.

5 Introduction to ATDD

This chapter is going to introduce ATDD and to discuss the goals behind it. The chapter is mostly based on Lasse Koskela's book "Test Driven, Practical TDD and Acceptance TDD for Java Developers" [15].

Acceptance Test-Driven Development is a software development technique which combines requirement specification with automatic executable tests of these requirements. There are several acceptance testing frameworks available. They subdivide onto following types: table-based, text-based and scripting language-based frameworks. Table-based frameworks are the most popular though and this work concentrates on a table-based automated testing tool FitNesse [18] which is going to be thoroughly reviewed in Chapter 9.

ATDD has three main objectives. First, it provides a shared communication medium to enhance exchange of information between all stakeholders, i.e. testers, developers and domain experts (people who provide software requirements). ATDD not only tries to make information flow easier between these groups but also *within* the groups. Second, ATDD provides instruments for storing functional software documentation which remains up-to-date throughout whole development phase. Finally, ATDD implies the system under construction constantly meets its requirements through automatic testing and remains in a good shape through constant refactoring.

This chapter is going to further describe the goals of ATDD, but first, the ATDD process should be uncovered.

5.1 ATDD process

ATDD is a test-driven technique, so a test needs to be implemented first. The term "test" can be very misleading though, especially when there nothing to test yet. It may also seem too abstract in the beginning and hence it may be beneficial (but not necessary) to first write a short description about the new test – a *user story*.

A user story, in context of ATDD, is a brief description of what exactly the upcoming acceptance test is going to test. The purpose of a user story is to catch the essential meaning of an acceptance test and to limit it from doing too much. User stories are meant to be easily and unambiguously understandable by domain ex-

perts and hence they are written in natural human language using domain terminology. User stories should be written by domain experts to get the most benefit out of the technique, but they still make tests easier to understand even if they were written by programmers.

Having a foundation for an acceptance test in form of a user story, the acceptance test itself is implemented. Acceptance tests normally set the system under test into a desired state, perform some actions on it and compare the results returned by the system to the expected ones. An acceptance test fails if the actual system outputs differ from the expected outputs and passes otherwise.

Running the test gives feedback about what work has to be done next. If the test fails, it should obviously be made pass. When it eventually passes, the inner design of the software under construction should be reviewed and improved. The test takes care about the feature under construction not getting broken — each modification should be followed by re-execution of the test. All other tests, which may be potentially affected by the changes, should be re-executed as well. When the programmer becomes satisfied with the design, the feature may be considered ready and another user story may be picked for processing. This makes the work flow cyclic and incremental.

The following sections are going to further uncover exactly which benefits ATDD brings into software development and what kinds of issues it addresses. ATDD is going to be compared to TDD quite often all along this work. People who are not familiar with ATDD and automated acceptance testing but who know something about TDD and unit testing usually have some strong preconceptions toward ATDD. This work tries to reveal the essential and unique properties of both techniques.

5.2 Shared communication medium

The motivation behind objectives presented in the beginning of this chapter concerns the human nature. Human beings are unable to communicate unambiguously and thus misunderstandings are especially common when trying to explain complex software logic.

In the beginning of a feature development its requirements are usually more or less abstract. Domain experts can provide some approximate sketch-ups but software developers need very concrete specifications in order to implement a program.

ATDD tries to eliminate ambiguity by presenting software requirements as examples of user actions being applied to the software. These examples describe how

exactly the system's states get changed from one state to another, i.e. which input values are required to initiate the state transition and which output values are going to validate the outcome. This approach relies on the basic idea that requirements expressed as concrete and objective examples with defined input and output values leave much less room for misunderstandings compared to abstract and subjective stories written in natural human language.

Acceptance tests help to lower the bar of handling tasks normally requiring certain competence and domain knowledge to all other team members but the few experts. Good tests don't require too much domain knowledge from developers and thus they are of great value for newcomers who need to learn the system. Normally a developer needs to know a system well in order to be able to maintain it. With help of acceptance tests it is possible to attach debugger to the software and to walk through the most complex execution paths learning the internals of the system. Such embedded documentation helps to detect and overcome issues with incomplete software requirements where parts of information are missing because they are considered obvious by domain experts but which can make requirements meaningless for developers.

5.3 Up-to-date documentation

Another common plague of software industry besides ambiguous specifications is the lack of up-to-date documentation. A big problem with software documentation is that it requires significant effort and discipline to keep in sync with the program code base. Program code gets constantly changed throughout development process but programmers are very unwilling to spend their time updating documentation. They'd rather write programs ([19][p.8] and [9][8th question]).

ATDD forces its practitioners to keep documentation up-to-date. This doesn't apply to all kinds of documentation (e.g. user manuals), but at least to the executable specification framework produced along the way of test-driving the application development.

Acceptance tests don't get out of date while they are *in use*. As long as development process obeys the rules of ATDD, meaning that no other code is being written than the code to make a failing test pass, the tests remain up to date. The keyword in "executable specifications" is *executable*. While traditional static documentation must be manually updated, executable specifications, by definition, get broken when they diverge from the system under test, i.e. when the system doesn't meet its requirements any more. They need to be manually updated too, but unlike

static documentation, executable specifications provide instant feedback about the system's state letting developers the possibility to fix the problem right after it has emerged and while it's still in their minds. If a new feature breaks some other acceptance tests than the feature's own tests, then either the feature itself or the broken tests need to be adjusted. In both cases the tests and the system evolve together to suite the new state of software.

Executable specifications have much better chance of not being abandoned and remaining up-to-date because writing them, unlike writing static non-executable documentation, is rewarding and even exciting, at least according to personal experience of the author of this text. Partially it's because running executable specifications lets instantly see what a program actually does whereas inspecting the source code only reveals what the program is supposed to do. A set of tests showing all green after a working day gives the wonderful feeling of control and self-confidence and provides some concrete evidence of one's job well done.

Executable specifications also provide much more documentation value for software developers than traditional paper documents. A programmer can actually start building the system relying on executable specifications because unlike informal documentation describing a system in ambiguous sentences, acceptance tests describe the system by examples with concrete input and output values.

Acceptance tests also help to *elicit* the concrete data values which are usually not explicitly stated by abstract user stories. The test interface created as a result of such elicitation improves overall testability and promotes loose coupling of software modules.

5.4 Automatic testing and maintenance

Executable specification framework can serve as functional documentation and knowledge storage which can be extremely helpful especially for less experienced software developers. It also improves their lives by making easier to set up the system into desired state during bug fixing activities.

Complex systems have a lot of interconnected modules many of which need to be set up before using even one of them. Without automatic test this means a lot of manual work. Executable specifications handle the dirty work automatically and let programmers concentrate on the essential — locating and fixing bugs. Additionally, a lot of knowledge must be available in order to run manual tests. In case of ATDD this knowledge is stored in executable specifications instead of domain experts' heads. This gives the word "automatic" two meanings: first, the tests get

executed by a computer program and not by a human making them much more time-effective. Second, there is no more need to ask domain experts for every little detail a developer doesn't know because most of domain logic is included in the automatic tests.

ATDD heavily relies upon automatic testing. The framework of automatic acceptance tests makes sure the system is never broken in many places at once. It lets software testers concentrate on their true job — trying to actually break the software — instead of spending their time desperately typing the same test values over and over again and hoping not to bump into one more bug which would mean one more exhausting series of test rounds.

The safety network consisting of automatic acceptance tests makes possible to refactor program code.

Martin Fowler [20] defines refactoring as follows:

> "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written."

Refactoring is like an investment making software maintenance easier. It doesn't affect the current functionality which is only a reflection of the current state of the software. The current state is however very likely to be just that — only a short moment in the life time of a software. As time goes by the software is anticipated to evolve and that's where refactoring plays a crucial role of ensuring the evolution is possible without pushing the software closer to chaos change by change.

Safe refactoring, however, is virtually impossible without proper testing. In Martin Fowler's (and others') book "Refactoring: Improving the Design of Existing Code" [20], he states the following:

> "If you want to refactor, the essential precondition is having solid tests."

Refactoring is also an essential part of the TDD process. Unfortunately, TDD is very hard to apply to legacy software which has not been developed according to TDD principles from the beginning and doesn't have a testable *internal structure*. That's where ATDD comes to help – executable acceptance tests access the *outer* interface of a software and thus can be applied even to legacy code. The following chapter is going to further clarify the differences between TDD and ATDD.

5.5 Summary of ATDD goals

To sum up, the main goals of acceptance test-driven development may be divided into three groups. First, acceptance tests serve as software requirements which automatically test software for meeting the requirements (or in other words executable specifications). Second, acceptance tests indirectly enhance source code quality by providing developers a security net giving the courage to modify source code lacking unit tests. Third, acceptance tests ensure there are no misunderstandings between requirement providers and developers. The following list represents the main goals of ATDD in a finer division.

- Acceptance tests serve as software requirements.
- Acceptance tests are implemented as executable specifications verifying that the system under test behaves as expected and satisfies the requirements.
- Acceptance tests serve as an unambiguous communication medium between software developers and domain experts who providing the requirements.
- Acceptance tests provide stakeholders a possibility to evaluate the software throughout all the development phase but not only when it is considered ready.
- Acceptance tests help to estimate the true progress status of a project and indicate when the software is ready.
- Acceptance tests make refactoring much easier by signalling when something gets broken during the refactoring process.
- Acceptance tests help developers unfamiliar with a software to explore it by providing executable use cases.
- Acceptance test-driven development leads to building software with good testability characteristics such as good API and loosely coupled modules.

5.6 ATDD pitfalls

ATDD promises many great things. There is one big precondition though. The development of software should indeed be driven by acceptance tests. Development of acceptance tests should not diverge from development of source code. The task is however easier said than done. Most projects involve rush phases when problems

need to be solved immediately and the urge to skip testing is very high. Doing so is dangerous concerning the ATDD process because writing source code not covered by acceptance tests affects not only the missing but the existing tests too.

According to the author's own experience, deviating from the ATDD process by writing untested code gives psychological permission to skip the whole testing for a while. Sure I'll write the tests later, one might think. However, after even a short period of programming without corresponding acceptance tests, many existing tests turn out to be broken without being noticed. This happens because software modules (especially concerning legacy systems) usually depend on each other and making change to one place may have unexpectedly broad consequences all over the system.

Now except the skipped tests for some new feature, also the existing broken tests need to be rewritten or fixed. Explaining the management why a considerable amount of time must be spent producing no program code at all might be hard especially when the budget is already tight.

Another big challenge with acceptance tests is organizing them. Even a small software has normally a lot of requirements which, in case of ATDD, mean a lot of acceptance tests. Test-driven development suggests a great effort needs to be taken to keep program source code clean and maintainable. The same applies to acceptance tests — as development goes by, the evolving acceptance tests need to be taken care of by removing duplication and restructuring separate tests to form a meaningful entity. Without paying additional attention to the shape of acceptance tests they will become a mess and will be eventually abandoned.

As it was noticed by the author of this text, organization of tests becomes much easier over time as certain development patterns evolve and the test framework becomes more comprehensive. The test written in the end of the project followed certain standards, they looked much cleaner (compared to those written in the beginning) while covering more functionality. A lot of different solutions had to be tried out before discovering the optimal one. Such a development of experience required quite a lot of time meaning that adoption of ATDD can turn out to be an expensive investment if there was no prior experience about it.

Finally, it's almost impossible to adopt ATDD without support and recognition from other team members. There is simply too much work to do keeping the growing acceptance test base under control solely by the effort of a few enthusiast programmers. In a software project conducted at the same time with the work on this thesis the author tried without success to invite domain experts to join working with executable specification. The possible reasons for unwillingness to adopt ATDD will

be reviewed later in Chapter 9.

The current chapter introduced the big goal of ATDD which is making software specifications automatically executable in order to automate the most part of unnecessary work normally done by programmers, testers and domain experts and let them concentrate on their real jobs.

ATDD has a close relative which has already been mentioned earlier – TDD. Although they sound almost the same, they have absolutely different goals which often causes a lot of confusion to learners and novice practitioners of the techniques. The following chapter is going to introduce TDD and discuss its essential purpose in order to further clarify the difference with respect to ATDD.

Note: there is some more criticism of ATDD supported by numerous case studies in Chapter 11.

6 Test-Driven Development

Kent Beck describes TDD in only a few sentences [21, p. vii]:

> ”In Test-Driven Development, you:
>
> - Write new code only if you first have a failing automated test.
> - Eliminate duplication.”

The two clauses together form, as Beck puts it, the TDD mantra — red/green/refactor. The description is short however these rules have broad consequences which will be discussed later in this chapter, but first, the most essential idea behind all test-*driven* techniques should be uncovered.

6.1 TDD is not about testing!

One of the most important things to realize about all test-driven techniques (TDD, ATDD and BDD) is that these are not *testing* techniques, regardless of the naming (a good article about the subject can be found from Esko Luontola’s blog [22]). Test-driven techniques imply creation of executable specifications prior to writing the actual program code which doesn’t make them testing techniques.

Such shift in thinking helps to understand the irrelevance of common questions like ”which features on the list should be tested and which shouldn’t” and ”what kind of features in general should be tested with TDD and by whom”. The answers become evident: TDD should be used by programmers as a technique for discovering the optimal solution for problems which are not obvious to solve.

At the time of creation of tests there is simply nothing to test yet. These ”tests” actually become *tests* only after domain functionality has been fully implemented. Only then they start to take care of the integrity of a system. Before that, the only purpose of ”tests” is to, first, help project members to unveil the true essence of the problem being solved, second, establish a shared understanding about the problem between domain experts and programmers and third, to help programmers in *discovering* how exactly code units under construction ”would like” to be structured.

Realizing the real nature of test-driven techniques also helps to figure out how they relate to other development techniques. TDD, as well as ATDD and BDD which

will be presented later, have not been designed to be a part of any kind of sequential process structure like ”first we plan, then we do some TDD and finally we start the integration testing”. TDD is an independent process and not a component of some bigger picture like in case of requirement elicitation or testing inside V-model. Although the true development models like XP or V-model can and should employ TDD, it cannot be compared to different types of testing because TDD *is not a testing activity*.

TDD is not a testing technique but rather a programming style involving writing ”assertions” about what the program is doing additionally to program code itself. As Dan North has pointed out in his article ”Introducing BDD” [23], the word ”test” is quite an unsuccessful term combined with ”driven” because it takes the reader's attention away from the essential ”driven” part and fills it with ”testing” which is normally conceived as a separate activity of validating program's behaviour taking place after a program or its part has been implemented.

Although ”specification” would be a better alternative for ”test”, both TDD and ATDD include the word in their names, so it just has to be used throughout this work.

The next section covers the TDD work-flow.

6.2 TDD work-flow

The work-flow according to TDD consists of three parts – writing a test, writing program code and refactoring.

Red and green are the colors used by most testing tools to indicate test failures and passes respectively. The three words – red, green and refactor – describe thereby the TDD process.

First, the test is written and run. The software is supposed to fail the test because the functionality being tested is not yet implemented by then. The first failing test ensures there is a need to make some modifications to the software. If the first test passes, then there is either something wrong with the test or the functionality is already implemented and the software doesn't need any more modifications.

Once having a failing test, the desired functionality is implemented by making the test pass. That means no other changes should be done than the changes absolutely required to pass the test. The constraint ”only ever write code to fix a failing test” is needed to avoid over-engineering by making code more complex than necessary. If some new functionality needs to be implemented, than it should first be backed up by a test. More to say, the code written to pass the test doesn't need to be

nice. The only purpose on the green stage is to make a test pass as quick as possible. Making code clean deserves its own separate process phase.

Finally, when program behaviour is being controlled by a group of passing tests, the program code is ready to be cleaned up. All the ”smells” like duplication, hard-coded variables, large methods and classes and other things making code ”dirty” and unmaintainable are removed and the code structure is re-organized, or briefly – refactored.

6.3 The goal of TDD

The overall goal of TDD is to build up the software in small cycles gradually adding functionality without breaking the existing code with each new increment. This is actually one of the main principles of agile software development – as it was mentioned earlier, agile software should evolve by small steps until it may be considered ready.

Assuming that tests are small enough, the process of writing only such code which is meant to pass some test leads to incremental nature of development and good decoupling. Apart from that, the test-first approach makes produced code testable by definition. As Koskela puts it [15, p. 17], ”Because we're writing the test first, we have no choice but to make the code testable.” Good testability ensures even a complex design is easy to refactor making code clean which in turn makes extending the code and addition of new features easier. Moreover, good testability means that units under test have effective and functional interfaces making the overall system design better.

TDD started to gain popularity along with Extreme Programming (an agile technique) in the early 2000's. Since then it became one of the best known software development practices and the abbreviation TDD is associated namely with unit test-driven development. The more concrete goal of TDD is thus producing ”Clean code that works” [21, ix]. Clean code not only looks nice and feels good to work with, but it is also relatively easy (and cheap) to maintain and extend. Finally, it works. The last argument is however controversial.

6.4 TDD pitfalls

Probably the biggest problem of unit testing and TDD relates to its test-first nature. The idea about writing tests when there is yet nothing to test is simply too irrational,

especially for novice programmers. Programmers who don't understand what TDD actually is about often have quite a strong attitude against TDD – they want to be programmers, not testers. A study conducted at the University of Kansas [25] has proved that mature developers are much more willing to accept TDD than novice programmers. They know from personal experience how ugly and complex program code can get and that special tools are required to keep it clean.

A tool developed at the University of Jyväskylä, ComTest [26], aims at making unit tests more easy to begin with. Its goal is to lower the threshold of adopting TDD by shifting test code from a separate location closer to program code. ComTest lets programmers to embed testing code directly into program comments so that the burden of switching between unit tests and the program is minimal. ComTest makes unit tests easier to write and maintain by improving readability and making them considerably shorter. This brings unit *tests* closer to being *specifications*. ComTest proved to be an effective utility in teaching software programming to university students who don't yet have the experience of drowning in code flood and thus don't have strong enough motivation to start learning TDD. ComTest supports Java, C++ and C#. An example of ComTest usage is going to be presented later in Chapter 7.

Unfortunately, even successful adoption of TDD doesn't guarantee project success because working code is not the same thing as working software. A program code is considered working if it passes the tests which again, speaking of TDD, are unit tests. Roy Osherove defines [27] unit tests as follows:

> "A unit test is a piece of a code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterward. If the assumptions turn out to be wrong, the unit test has failed. A "unit" is a method or function."

The source code-related nature of unit test-based TDD makes it incapable of aiding the communication between developers and domain experts. If a developer misunderstood the software requirements or the requirements were otherwise invalid, then there is no way to prevent the flaw by means of TDD. All the tests would still happily signal green and the code would still "work".

Two other major issues concerning TDD are multi-threaded code and legacy code. While multi-threaded code is relatively rare, the latter is an obstacle on the legacy software's way to agile development.

In terms of TDD any code not covered by automated tests is legacy code. Adopting TDD in a legacy software is hard because even new code cannot usually be easily

tested by automated tests. When a new feature is being added to a legacy system, the new code has to use the existing components, that is to depend on them. Testing the new code with automated tests would thereby mean initializing and using legacy code objects which is often tricky without automated tests testing legacy code directly. Legacy code doesn't have the needed interfaces and hence it's not testable because it wasn't originally meant to be used by outsider-code such as unit tests.

External resources such as databases, network services and file systems add even more uncertainty to a successful adoption of unit test-based TDD especially when these are combined with legacy code. TDD is quite a mature concept and some kinds of solutions (such as mocks [24]) have certainly been found to overcome most of the problems. Still, adopting TDD in a large legacy project heavily based on a database and network services with lots of bad, unmaintainable and interdependent code would probably bring much less value if any comparing to costs.

The problems mentioned above are easier to solve with another kind of testing — the Acceptance Testing. While Test-Driven Development aims at producing clean working code relying on unit tests, Acceptance Test-Driven Development ensures the software does what it is supposed to do with help of acceptance tests. Following ATDD, software is tested on a higher level than with TDD. ATDD doesn't test each code unit directly but rather through some outer interface making ATDD suitable for testing legacy software too. The following chapter describes the technique in greater detail.

6.5 Acceptance testing opposed to unit testing

TDD seems to have exactly the same work-flow as ATDD which can be very confusing in the beginning, especially because nearly all the books and articles about ATDD urge to apply both techniques at the same time. Wouldn't it be a waste of capacity?

The short answer is – no. The longer answer is – it depends.

ATDD and TDD differ from each other mostly by their goals. TDD is all about code — every line of code is written only to satisfy a unit test. A code base growing that way is never broken in many places, it's always only a few changes away from tests to pass. If it isn't, then something other than proper TDD is being practiced. The main goal of TDD is to keep redundancy, unnecessary complexity and over-engineering out of code, in other words – to keep it clean and working. TDD is a technique for working with *program code* which makes it useless for anybody besides developers and architects, i.e. people who deal with code.

ATDD in turn is more of a link between domain experts (usually customers) and developers of software. It makes sure both parties have a common understanding about the problem being solved. The customer is seldom interested in code as such and maybe even more seldom capable of evaluating it, but instead, the customer is very interested in what the code actually does. ATDD provides domain experts the access to business logic through a comprehensible interface during the construction stage. Such an access gives system owners the possibility to prevent misunderstandings in the early development stage when fixing flaws is yet cheap.

The questions like "which features should be tested with TDD and which with ATDD" are very common yet somewhat misleading. Robert C. Martin, one of the creators of the Agile Manifesto, wrote a blog article [28] about the difference between unit and acceptance testing. The article suggests another way of thinking – not which technique is responsible of handling which kinds of situations, but *who are the testers*:

> "It's true that the two streams of tests test the same things. Indeed, that's the point. Unit tests are written by programmers to ensure that the code does what they intend it to do. Acceptance tests are written by business people (and QA) to make sure the code does what they intend it to do. The two together make sure that the business people and programmers intend the same thing."

Unit and acceptance testing operate on different abstraction levels and aim at different goals. In its essence, a unit test is a description of *how* exactly a feature is implemented in the code. An acceptance test in turn describes *what* a feature actually is.

Unit and acceptance tests may also be distinguished by different kinds of documentation value they both provide.

Unit tests may be compared to comments in source code as they describe *how* certain code units function, which variables they take in and return and what is the expected way of using certain code units. Unit tests serve as a guide for *software developers* in exploring and maintaining the source code.

Acceptance tests are in turn comparable to general program documentation specifying *what* the system's features are and targeting *domain experts* who provide the system requirements.

Unit and acceptance testing also differ from structural point of view.

As mentioned earlier, unit tests are strongly tied to source code. Moreover, as Michael Feathers presented at SD West 2007 conference (presentation slides [29]),

unit testing is a process of ”testing a (small) portion of software independently of everything else”. He adds even more restrictions to the concept of unit tests:

> ”A test is not a unit test if:
>
> 1. It talks to the database
> 2. It communicates across the network
> 3. It touches the file system
> 4. It can’t run correctly at the same time as any of your other unit tests
> 5. You have to do special things to your environment (such as editing configuration files) to run it.”

The restrictions written above define unit tests as fast to run and as easy to implement as possible. These are good attributes for any kind of tests, however, they are vital for unit tests. As the name suggests, each unit test should test a single object and not groups of them, which would actually happen if any of the restrictions listed above were broken.

On the technical level ATDD does exactly the opposite: it tests *groups* of units. Execution of an acceptance test starts by consuming one or several API methods expecting particular results. Based on the results, the test either passes or fails, but unlike unit tests, acceptance tests may consume many units at a time and test how they ”co-operate” as a group to provide the expected results.

Acceptance tests (unlike unit tests) may also involve communication with external resources like database and file system. This kind of testing might remind of integration testing, however, ATDD has different goals. For example, unlike ATDD, integration testing is not meant to serve as executable specification.

Finally, unit and acceptance testing differ by types of requirement sources. Acceptance tests satisfy requirements stated by user stories. In turn, the source of requirements in TDD is not a user story but other code units utilizing the target code being tested.

To sum up, TDD and unit testing get utilized by programmers for making program code better while ATDD and acceptance testing help both programmers and domain experts to gain a common understanding about the problem being solved. Answering the initial question whether it would be a waste of capacity to practice TDD and ATDD simultaneously – it depends on the project, which one of the techniques is more beneficial. If TDD has been applied from the beginning of a project and the software is widely covered with unit tests, it might be very rational to use

both techniques. However, if the project is a "legacy project" (i.e. it doesn't have any unit tests, it is not internally testable), it will most likely be too expensive to bring TDD in. Such projects are a home ground for ATDD – legacy code is where ATDD shines the most.

The following chapters give some usage examples of two common automated testing frameworks — NUnit [30] and FitNesse. The former is a unit testing framework for .NET and the latter is an ATDD framework.

7 Unit testing with NUnit

This chapter is going to represent unit testing with a working example of testing a calculation algorithm. Later, Chapter 9 is going to walk through a working acceptance test. The two tests together are meant not only to describe the technical details but also to further review the different goals of TDD (the unit test) and ATDD (the acceptance test).

It's worth mentioning that the purpose of this chapter is to expose the true nature of unit testing as a technique for creation, improvement and evaluation of program *source code*, the part of software normally visible only to developers.

The algorithm which is going to be described later in this chapter was actually first tried to be developed with ATDD. However, it was definitely a wrong decision as ATDD normally operates at an external system interface and thus it doesn't provide much of help in designing the actual program code. Relying solely on ATDD during code construction lets too much space for all those bells and whistles programmers consider cool during construction phase but which often turn out to be useless and confusing in the final design. A unit test, in turn, is supposed to *directly* access the interface of a code unit (a method) under construction forcing a programmer to focus on the essential — exactly which input values are available and exactly which output values are desired from certain code units.

Finally, proper unit tests take much less time to execute compared to acceptance tests which is an especially important quality from programmers' point of view because discovering the optimal solution for a complex problem may require hundreds of test rounds. Acceptance tests may take up to several minutes to finish which is a way too long time to wait in the middle of programming process.

This and the following chapters include some terms from financial sector which most people are unfamiliar with. These terms are however absolutely irrelevant in the context of TDD, ATDD or any other development technique described in this work. The author had tried to explain the terms, but it turned out to be a hard task because explaining one term brings in ten other terms to be explained. Financial sector has an extremely complex domain logic which usually doesn't have any linkage to the world familiar to most people. The author still wanted to include some real-life examples to make examples more natural, but deepening into details would grab the reader's attention away from the essential part — the technique being dis-

cussed. Thereby only the most important terms are going to be briefly uncovered along the way.

NUnit is an open source unit testing framework supporting all Microsoft .NET languages. It belongs to a family of xUnit testing frameworks. Some other members of xUnit family are, for example, JUnit for Java language, CppUnit for C++ and PyUnit for Python. There is a unit testing framework for nearly any language (list of frameworks in Wikipedia [31]).

An NUnit test is a program method taking no parameters and returning no values. A test passes if no exceptions were thrown and fails otherwise. Exceptions are not normally thrown directly (i.e. with a ”new” operator) but with NUnit's Assert-methods. A unit test example is going to be presented later in this chapter.

7.1 Requirement elicitation

In order to write unit tests the requirements need to be elicited first. *However*, it's not necessary to understand the domain in order to follow this chapter because the purpose of the following problem is only to describe how exactly TDD is utilized to translate domain problems into solutions in form of program code.

The problem domain includes a series of financial instrument *lots* – purchase lots. A purchase lot is an entity containing a defined amount of a financial instrument (e.g. two lots of 10 shares of Nokia totalling 20 shares or three lots of 5.25 ounces of Gold totalling 15.75 ounces). Instrument *fraction* denotes how many decimals certain instrument can have. For example, there might be three purchase lots (fraction = 4): 30.1671, 63.7725 and 10.0012 pieces respectively. In a specific corporate action (CA, a financial event issued by a company) new shares are detached from existing ones. In case of ”1:1” detachment each source purchase lot would produce one target lot of equal amount. However, the degree of complexity grows when source and target instruments can have different fractions and the detachment relationship may have up to nine decimals (i.e. 1.123456789:4.327 instead of 1:1). Moreover, there are several calculation rules to make a detachment. According to one calculation method (DirectTruncate) and given that fraction of target security is two decimals, the first lot of 30.1671 pieces produces a target lot of 20.11 pieces (3:2 detachment):

```
Truncate(30.1671 / (3/2), 2) = 20.11
```

However, according to another calculation method (TruncateTruncate), only 20 pieces are detached:

```
Truncate(Truncate(30.1671/3, 2) * 2, 2) = 20.00.
```

Additionally to different calculation rules (there are six of them) there are some more requirements; both source and target instruments should not be wasted in calculations which happens with truncation. In case of TruncateTruncate there are 0.0171 pieces of the **source** lot wasted:

```
30.1671 - Truncate(30.1671/3, 2) * 3 = 0.0171
```

The remaining part should be taken into account when calculating the detachment amount of the following source lot. The source of detachment from the second lot should not be the original amount of 63.7725 pieces but 63.7896 pieces instead (63.7725 + 0.0171). The same principle applies to detachment target.

To make long story short, there is a list of requirements needed to be satisfied by the program and validated with unit tests. However, as stated earlier, particular requirements (and understanding of them) are not relevant in this case. The purpose of this chapter is to describe what kind of problems in general unit testing is meant to solve. Let's start with the tests.

7.2 Example unit test

The following listing shows a test class containing a unit test for each of the six calculation rules.

```
[TestFixture]
public class DTCalcTest : DTCalcTestBase {

    private decimal a, b;
    private int sf, tf;

    [SetUp]
    public void SetUp() {
        sourceLots = new decimal[] { 222.6M, 239.2M, 743.8M };
        a = 8.0987M;
        b = 2.8323M;
        sf = 1;
        tf = 2;
    }

    [Test]
    public void DirectRound() {
        dtCalc = new DTCalculator(a, b, sf, tf, CalculationType.DR);
        Assert.AreEqual(new decimal[] { 77.85M, 83.65M, 260.13M },
            CalculateTargetAmounts());
    }
```

```
    [Test]
    public void DirectTruncate() {
        dtCalc = new DTCalculator(a, b, sf, tf, CalculationType.DT);
        Assert.AreEqual(new decimal[] { 77.84M, 83.66M, 260.12M },
            CalculateTargetAmounts());
    }

    [Test]
    public void RoundRound() {
        dtCalc = new DTCalculator(a, b, sf, tf, CalculationType.RR);
        Assert.AreEqual(new decimal[] { 77.89M, 83.55M, 260.29M },
            CalculateTargetAmounts());
    }

    [Test]
    public void RoundTruncate() {
        dtCalc = new DTCalculator(a, b, sf, tf, CalculationType.RT);
        Assert.AreEqual(new decimal[] { 77.88M, 83.56M, 260.28M },
            CalculateTargetAmounts());
    }

    [Test]
    public void TruncateRound() {
        dtCalc = new DTCalculator(a, b, sf, tf, CalculationType.TR);
        Assert.AreEqual(new decimal[] { 77.61M, 83.83M, 260.01M },
            CalculateTargetAmounts());
    }

    [Test]
    public void TruncateTruncate() {
        dtCalc = new DTCalculator(a, b, sf, tf, CalculationType.TT);
        Assert.AreEqual(new decimal[] { 77.6M, 83.84M, 260M },
            CalculateTargetAmounts());
    }
}
```

Listing 7.1: DTCalcTest class

A class containing test methods (a so-called test fixture) has the *TestFixture* attribute. Test methods themselves are marked with the *Test* attribute enabling NUnit framework to "see" tests.

The SetUp method is run by NUnit before each test. It initializes the input values (222.6M, 239.2M, 743.8M) which are going to be used by all six test methods.

Each test method applies a certain type of calculation (DR, DT, etc.) and compares expected values (the first argument of Assert.AreEqual) to the actual result returned by CalculateTargetAmounts.

The method *CalculateTargetAmounts* follows:

```
using NUnit.Framework;
using System.Collections.Generic;
using System;

public abstract class DTCalcTestBase {

    protected decimal[] sourceLots;
    protected DTCalculator dtCalc;

    protected decimal[] CalculateTargetAmounts() {

        List<decimal> targetAmounts = new List<decimal>();
        decimal netTarget;

        foreach (decimal sourceAmount in sourceLots) {

            netTarget = dtCalc.ProcessNextSource(sourceAmount);

            if (netTarget > 0)
                targetAmounts.Add(netTarget);
        }
        return targetAmounts.ToArray();
    }
}
```

Listing 7.2: DTCalcTestBase class

CalculateTargetAmounts defines how exactly the unit being tested (DTCalculator) is designed to be used.

Finally, the actual program code (the system under test) is presented for the sake of clarity.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

public enum CalculationType { DR, DT, RR, RT, TR, TT };

public class DTCalculator {

    private decimal sa;
    private decimal ta;

    private decimal a;
    private decimal b;
    private int sourceFraction;
    private int targetFraction;
    private CalculationType calcType;

    /// <summary>
    /// Calculator of detachment amount.
    /// </summary>
```

```
/// <param name="a">Detachment source block amount (split)</param>
/// <param name="b">Detachment target block amount (split2)</param>
/// <param name="sf">Source fraction </param>
/// <param name="tf">Target fraction </param>
/// <param name="ct">Calculation type</param>
public DTCalculator(decimal a, decimal b, int sf, int tf, CalculationType ct){

    this.a = a;
    this.b = b;

    if (a == 0 || b == 0) {
        throw new ApplicationException(
            "Split values cannot be zero");
    }

    sourceFraction = sf;
    targetFraction = tf;
    calcType = ct;
}

/// <summary>
/// Calculates the detached amount according to source
/// lot amount, amount calculation type and split values.
/// </summary>
/// <param name="nextSa">Amount of next source lot</param>
/// <returns>Detached amount</returns>
public decimal ProcessNextSource(decimal nextSa) {

    decimal nt; //net target amount
    decimal gt; //gross target amount

    decimal dct;
    decimal tct;
    decimal rnd;

    if (nextSa < 0) {
        //short lots must be converted
        //before calling ProcessNextSource
        throw new ApplicationException(
            "Negative source amount: " + nextSa.ToString());
    }

    sa += nextSa;

    if (sa < 0) {
        //too small nextSa
        //while sa < 0
        return 0;
    }

    try {

        dct = (sa / a) * b;
        tct = Truncate(sa / a, sourceFraction) * b;
        rnd = Round(sa / a, sourceFraction) * b;
```

```
switch (calcType) {
    case CalculationType.DR:
        gt = dct;
        nt = Round(gt + ta, targetFraction);
        break;
    case CalculationType.DT:
        gt = dct;
        nt = Truncate(gt + ta, targetFraction);
        break;
    case CalculationType.RR:
        gt = rnd;
        nt = Round(gt + ta, targetFraction);
        break;
    case CalculationType.RT:
        gt = rnd;
        nt = Truncate(gt + ta, targetFraction);
        break;
    case CalculationType.TR:
        gt = tct;
        nt = Round(gt + ta, targetFraction);
        break;
    case CalculationType.TT:
        gt = tct;
        nt = Truncate(gt + ta, targetFraction);
        break;
    default:
        throw new ApplicationException(
            "invalid calcType: "
            + calcType.ToString());
}

if (gt < 0) {
    //should not be possible
    throw new ApplicationException(
        "gt < 0: " + gt.ToString());
}

if (nt < 0) {
    //neg. nt; sa -> ta conversion
    //should not yet happen
    return 0;
}

//source amount gets converted
//into target amount while some
//part (nt) is going away
sa -= gt * (a / b);
ta += gt - nt;

//if nt = 0 then there is no point in calling
//ProcessNextSource(0) with even less sa
if (nt > 0 && sa > 0) {
    //sa may be enough for some more targets
    return Round(nt + ProcessNextSource(0), 6);
}
```

```
            return Round(nt, 6);

        } catch (Exception ex) {
            throw new ApplicationException(
                "System error: CalculateNextTargetAmount failed", ex);
        }
    }

    //AwayFromZero = Regular rounding (vs. Bankers' rounding)
    public static decimal Round(decimal d, int fraction) {
        return Math.Round(d, fraction, MidpointRounding.AwayFromZero);
    }

    //Trancates to a given fraction
    public static decimal Truncate(decimal d, int fraction) {

        d = d * (decimal)Math.Pow(10, fraction); //34.56 -> 345.6 (fraction = 1)
        d = Math.Truncate(d) / (decimal)Math.Pow(10, fraction); //345 -> 34.5

        return d;
    }
}
```

Listing 7.3: System under test

Unit tests are meant to test code units while unit testing frameworks provide a convenient way of composing and running those tests. Sometimes, however, tests become messier than they could be. The fixture class DTCalcTest presented earlier has six tests which have only a couple of different values. The rest of the code is duplication. A tool mentioned earlier, ComTest [26], comes for help:

```
/// <summary>
/// Calculates the detached amount according to source
/// lot amount, amount calculation type and split values.
/// </summary>
/// <param name="nextSa">Amount of next source lot</param>
/// <returns>Detached amount</returns>
/// <example>
/// <pre name="test">
///  DTCalculator cal;
///
///  cal = new DTCalculator(8.0987M, 2.8323M, 1, 2, CalculationType.$calcType);
///
///  cal.ProcessNextSource(222.6M) === $r1;
///  cal.ProcessNextSource(239.2M) === $r2;
///  cal.ProcessNextSource(743.8M) === $r3;
///
///   $calcType | $r1    | $r2    | $r3
/// ---------------------------------------
///     DR      | 77.85M | 83.65M | 260.13M
///     DT      | 77.84M | 83.66M | 260.12M
///     RR      | 77.89M | 83.55M | 260.29M
///     RT      | 77.88M | 83.56M | 260.28M
///     TR      | 77.61M | 83.83M | 260.01M
///     TT      | 77.6M  | 83.84M | 260M
/// ---------------------------------------
///
/// </pre>
/// </example>
public decimal ProcessNextSource(decimal nextSa){

  decimal nt; //net target amount
    ecimal gt; //gross target amount

  Other contents (omitted)...
}
```

Listing 7.4: ComTest example

The tabular test presented above executes exactly the same tests described earlier, but in a lot more compact form. Tabular form appears to be a very efficient and clean form to test multiple data values. It makes possible to completely remove duplication.

Another goal of ComTest is to bring tests closer to program code so that programmers have a lower barrier to write tests — this way there is no need to switch between different files (domain and test code). Both program and test code can be edited at the same time. The vicinity of unit tests also improves code documentation by becoming a part of it. Moreover, code documentation becomes less inclined to get outdated by including executable tests. If domain functionality got modified on purpose, the tests will remind to update the documentation too.

This chapter presented an example of a "proper" unit test as discussed in Chapter 6.5, meaning that the test doesn't access external resources like databases and doesn't depend on any other unit tests which enables it to run concurrently.

However, the truly fundamental idea behind unit testing is not that much about efficiency of execution, but about the principle of "divide and conquer". Unit testing urges to encapsulate features (or algorithms like in this chapter) into *independent code units* in order to reveal a programmer the actual essence of a domain problem and to help in finding the optimal implementation.

The following chapter is going to introduce BDD, a development technique aiming at driving software development with behaviour specifications. Unlike TDD, it doesn't try to describe internal structure and directly improve program code but rather guides development of *software that matters*. The following chapter is meant to serve as a link between unit testing and table-based acceptance testing.

8 Behaviour-Driven Development

BDD was introduced in 2006 by Dan North [23]. Dan North gave the following definition of BDD in 2009 in a podcast [32]:

> ”BDD is a second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology. It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software that matters.”

Despite of a rather vague definition, the main principles of BDD are simple.

8.1 The motivation behind BDD

Initially, the basic motivation behind BDD was the difficulty of adoption of TDD by programmers not familiar with it. TDD is meant to test-drive creation of software code, i.e. to test such code which doesn't exist at the moment of creation of unit tests. This kind of approach turned out to be hard to assimilate for newcomers. North mentions some common questions he's been constantly asked by programmers: where to start, what to test and what not to test, how much to test in one go, what to call their tests, and how to understand why a test fails.

These questions relate to the abstract nature of unit tests — they are supposed to initiate the implementation of something (program code), the requirements of which are not clearly defined. The word ”requirements” here means the requirements of particular program code units and not software requirements. North felt there needed to be a more concrete layer describing domain logic than unit tests. This layer had to be based on the actual needs of domain logic and not on presumable needs of program implementation units. North wanted to shift the balance between code implementation and software requirements away from program code toward domain logic. He also wanted to get rid of the word ”test” in Test-Driven Development because it tightly associates with testing, a post-construction activity, and hence misleads TDD newcomers. These artifacts which drive development are not actually tests but examples of program behaviour, at least at the moment they are being created.

8.2 Behaviour over implementation

Thereby, North proposed the term Behaviour-Driven Development and introduced the first BDD tool, JBehave (project web page [33]), which was written by himself. JBehave emphasizes behaviour over testing by exercising a certain vocabulary. BDD specifications normally follow the Give-When-Then (GWT) notation:

```
Given the system under test is in state A
When event X occurs
Then the system's state becomes B
```

Listing 8.1: GWT notation

Such kind of test automatically makes irrelevant those questions expressed earlier (where to start, what to test and so on). A problem of "how to implement the program" transforms into a problem of "what exactly needs to be achieved by the program". This is precisely the same problem ATDD intends to solve. BDD only has a better name which makes prominent the fact of BDD not being a testing activity.

Neel Narayan mentions the common problems arising upon adoption of BDD in his blog article "BDD is not about tools" [34]. People learning BDD often try to find special tools to drive the process. They ask questions about what BDD tool they should use, how exactly that tool should be used, which tool is better than others etc. The article, however, states that using a tool doesn't necessarily mean practicing BDD and that paying too much attention to technical details leads to losing the true value of BDD.

Compared to ATDD, which is usually based on table tests, BDD is fairly tough in emphasizing presentation simplicity and the importance of *behaviour*. BDD aims at capturing software requirements as a list of "behaviour specifications", i.e. statements about what the software should do. According to Narayan, each specification has two components: Narrative and Scenario parts. The Scenario part consists of a programmatically executable set of instructions expressed in GWT notation presented earlier. The Narrative part is meant to clarity the reasons behind certain functionality. It explicitly defines *who* is requesting certain behaviour and *why*. The following example includes both Narrative and Scenario parts:

```
As an efficient software developer (role),
I need to run a suite of unit tests fast enough (feature),
so that I can afford running them often enough (benefit).

Given a suite of 500 average unit tests
When the suite gets executed
Then the execution should take less than 1 minute.
```

Listing 8.2: Behaviour reasoning

Together these two parts form a vivid description of what the software should do in context of business value. The listing above looks much like a user story — it is short and easy to understand but still includes the essential part of the feature. The only difference — it's executable.

The example above seems straightforward but would actually contain quite a lot of program code to implement it. And that is in fact an inherent goal of BDD — to hide implementation complexity and reveal only the essential parts of requirements. Although BDD may be practiced even without any tools (blog article [34]), popular BDD tools like JBehave, Cucumber and Concordion (project web pages [33], [35], [36]), all emphasize the importance of behaviour and test readability. Unfortunately, complex behaviour is not the only thing making software complex.

8.3 Complex behaviour doesn't come alone

"Hello, World!" examples which are usually presented in getting started tutorials always have only a couple of data values, likewise those examples demonstrating the importance of behaviour. The real world is different though. Additionally to complex behaviour there are complex data structures with vast amounts of information. Some tools suggest hiding such data in fixtures (i.e. program code) [36] which doesn't actually serve the BDD tools' purpose of a communication medium. In order to communicate efficiently, both kinds of stakeholders — domain experts and software developers — need to have access to both behaviour logic and the data involved in that behaviour.

The bare Given-When-Then notation is efficient in handling behaviour but not data. Consider the following example:

```
Given a password validation algorithm demanding a number
When the password "PassWord" is being created
Then the system should reject the password:
  "A password must contain a number".
```

Listing 8.3: Password must contain a number

The algorithm also validates password length:

```
Given a password validation algorithm demanding at least
  8 characters
When the password "P4ssWrd" is being created
Then the system should reject the password:
  "A password must be at least 8 characters long".
```

Listing 8.4: Password must be long enough

What about upper- and lowercase characters, dictionary words and other possible requirements? They all need to be written down as separate cases which creates unnecessary duplication and adds up accidental complexity (see Chapter 2).

To address this issue, some BDD tools (at least JBehave and Cucumber) provide support for tabular parameters. The following example validates password algorithm in a different way:

```
Given a password validation algorithm
When a user provides a new password
Then the system should react as follows:
|Password|Message                                  |
|PassWord|a password must contain a number         |
|4ssWord |a password must be at least 8 characters long|
|p4ssword|a password must contain uppercase letters |
|P@ssw0rD|the password is accepted                 |
```

Listing 8.5: JBehave tabular parameters

The example presented above is meant to describe the inefficiency of plain Given-When-Then notation concerning multiple variables. However, it's still a hello-world example having nothing to do with reality. Real-life scenarios not only validate some domain logic but also need to prepare a system under test for a test case. A more realistic scenario would hence look somewhat like the following:

```
Given a customer:
|CustomerCode|CustomerName|Country|Currency|
|6538764     |John Smith  |USA    |USD     |
And a couple of portfolios:
|CustomerCode|PortfolioCode|PortfolioType  |
|6538764     |FIFO_1       |FIFO           |
|6538764     |AVG_1        |Average priced|
And a security:
|SecurityID|SecurityName|Currency|Decimals|
|NOK       |Nokia       |USD     |2       |
And some purchase lots in both portfolios:
|TransactionType|SecurityID|Amount|Portfolio      |
|Purchase       |NOK       |172.12|6538764/FIFO_1|
|Sale           |NOK       |83.47 |6538764/FIFO_1|
|Purchase       |NOK       |100.00|6538764/AVG_1 |
When the current position for customer "6538764" is requested
Then the system should display:
|CustomerCode|PortfolioCode|SecurityID|Amount|
|6538764     |FIFO_1       |NOK       |88.65 |
|6538764     |AVG_1        |NOK       |100.00|
```

Listing 8.6: The chaos of tables

The example above is still an imaginary illustration because a lot of information is missing: currency rates, security prices, transaction dates, brokerage fees and so on. In fact, the real tests the author of this text had to deal with are ten to fifty times

bigger than the one presented here. As one might point out, such data-richness and logic complexity are not typical of all software. Unfortunately, many systems are indeed that complex and the more complex a system appears to be, the more desperately it longs for help in managing its complexity.

Despite of a somewhat cropped version of a real test case, the tendency starts to get unfolded — the tables start growing, the variety and volume of data increases and the whole test becomes more complex, error-prone and harder to follow. As a consequence, the ability to manage data tables efficiently becomes essential.

The last sentence leads to the most important finding of this work — in order to succeed in test-driving development of logically complex and data-rich applications, the test data must be sufficiently easy to manage and refactor. Otherwise the test tables simply drown in their complexity, get outdated and finally become abandoned. Another sad finding is that there seems to be no tools existing which provided such a functionality, at least considering the open-source sector.

The following chapter will have more discussion about the matter of data complexity and its management. It is going to describe usage of FitNesse, a table-based acceptance testing tool. Unlike BDD tools, FitNesse doesn't have any mandatory notation forms but gives users the flexibility to decide how exactly the tests should be structured.

9 Acceptance testing with FitNesse

FitNesse is an open-source acceptance testing tool. It can execute tests written in the following languages: Java, all .NET languages (including C# and VB.NET), C++, Delphi, Python, Ruby, Smalltalk and Perl.

FitNesse works in a slightly different way compared to NUnit. NUnit is based on exceptions — test code gets executed and the test passes unless an exception is thrown either directly with the ”new” operator or by a failing assertion.

FitNesse in turn relies on communication with the SUT (system under test). Roughly speaking, FitNesse executes a SUT method with user-provided input values and compares output values supplied by the SUT to the expected ones which are also provided by the user. A test fails if any of the expected value differ from the actual SUT output. A test also fails on exceptions (with yellow coloring) but this is not the expected behaviour – the SUT is always supposed to return its results.

Before diving into syntax details and usage examples, the actual interface accessed by tests should be discussed.

9.1 ATDD vs GUI testing

From a domain expert's point of view, probably the best way to test a program is to use a program exactly the same way it would be done during its normal operation. Such strategy is known as record/playback method and there are plenty of tools supporting it. In a nut shell, the user presses the record button of a testing tool, then executes a series of actions being tested and stops the recording. The tool records all the user actions and can re-play them later. It also compares the actual results to expected ones. The technique executes all the code involved in certain user action — starting from the user interface, passing all the domain logic to database and back again to the UI — gets executed. It also provides domain experts an intuitive way of expressing executable specifications (i.e. acceptance tests) because they are built upon familiar user interface.

The problem with such approach is that a new feature being introduced does not necessarily have a user interface yet which disagrees with test-driven methodology demanding an executable specification to be implemented before the feature. A missing user interface might not be a big deal if it was a computer-accessible API

like a SOAP web service or a text-based command prompt — this kind of interfaces involves much less ambiguity and hence is relatively easy to build. Graphical User Interfaces (GUIs) are different. Robert C. Martin (A.K.A. Uncle Bob, one of the authors of the Agile Manifesto) wrote in his posting about test-driving GUI on Stack Overflow [37]:

> "Test driving UI is problematic because you often don't know what you want on the screen until you see it on the screen. For that reason, GUI development tends to be massively iterative and therefore very difficult to drive with tests."

Also Alex Ruiz and Yvonne Wang Price from Oracle discuss the difficulties of test-driving GUI in their article "Test-Driven GUI Development with TestNG and Abbot" [38]. The article tells about GUIs being designed for humans which makes them hard to test-drive. GUIs respond to user-generated events which need to be simulated and handled by testing tools. Writing such tools can be tedious and error-prone because the room for potential interactions with GUI is huge — user can press any button at any time and consume any unrelated features. The authors further mention the two common GUI testing methods — the record/playback method and programmatic GUI tests. Again, the record/playback method is not suitable because there is no GUI existing when a feature is being designed and thus there is nothing to record. The authors propose programmatic GUI tests as a solution and suggest two tools to be used: jfcUnit and the Abbot Java GUI Test Framework (project web pages [39], [40]). These tools let developers access Java Swing GUI objects and execute the needed events on them. Unfortunately these tools haven't passed the test of time — jfcUnit is completely dead by now (2011) and the Abbot had released over three years ago last time and its mailing list is quite empty too.

The reason is fairly obvious: these tools still depend on GUI components too much. Even though they let programmers implement tests in advance instead of post-construction record/playback testing, the necessity of GUI control manipulations brings too much accidental complexity (see Chapter 2).

Most ATDD gurus (e.g. Robert C. Martin, Michael Feathers, Lasse Koskela) recommend to completely exclude the GUI layer from acceptance tests and concentrate on the actual domain logic instead. Michael Feathers wrote about the subject in his article "The Humble Dialog Box" [41].

Lasse Koskela describes two main strategies of accessing the software under test which both skip the GUI layer: "crawling under the skin" and "exercising the internals" [15][p.404]. The former means attaching acceptance tests exactly below the

GUI layer while the latter denotes executing test cases directly against program code implementing the features under test. These two techniques are going to be further explained next.

Figure 9.1 illustrates a 3-tier application consisting of client and server applications and a database. All communication between these three layers happens over network connections. The client consists of a graphical user interface (meaning the actual GUI controls like buttons and text fields) and a separate level of program code which serves as a link between GUI controls and the server application. The server in turn contains the actual domain logic indicated by ″Customer services″ and ″Transaction services″ boxes.

According to ″crawling under the skin″ the application needs to be implemented in such a way that only minimal portions of programs code reside in immediate vicinity of GUI controls and as much as possible should be moved into separate classes. For example, GUI event handlers should never call methods of server application directly but they should call the additional ″proxy layer″ first (marked with the upper red arrow in Figure 9.1) and only that additional layer should process all the communication with server. The additional layer in client application is meant to be called by both GUI controls and acceptance testing framework. It makes possible to postpone development of GUI to a very late phase of project until most of domain logic is already done and tested and thus there is much less uncertainty about the GUI.

″Crawling under the skin″ covers almost as much code as record/replay method because almost all the client code (concerning the feature under test) gets executed in a test as well as all the server code. However, there is a big drawback — test execution speed suffers because there is at least one network layer between client and server. Depending on whether there is a lot of complex and important client code to be tested or not, it might be rational to completely skip client testing.

″Exercising the internals″ strikes directly where the problem (or feature) is. It skips the additional network connection between client and server and executes tests directly against server code making tests up to an order of magnitude faster [15][p.404]. This method skips all the client code leaving it completely untested, but results in significant performance boost which may be crucial in case of hundreds or thousands of tests.

Later in this chapter there will be an example acceptance test implemented according to ″exercising the internals″ strategy, but first, the very basics will be described.

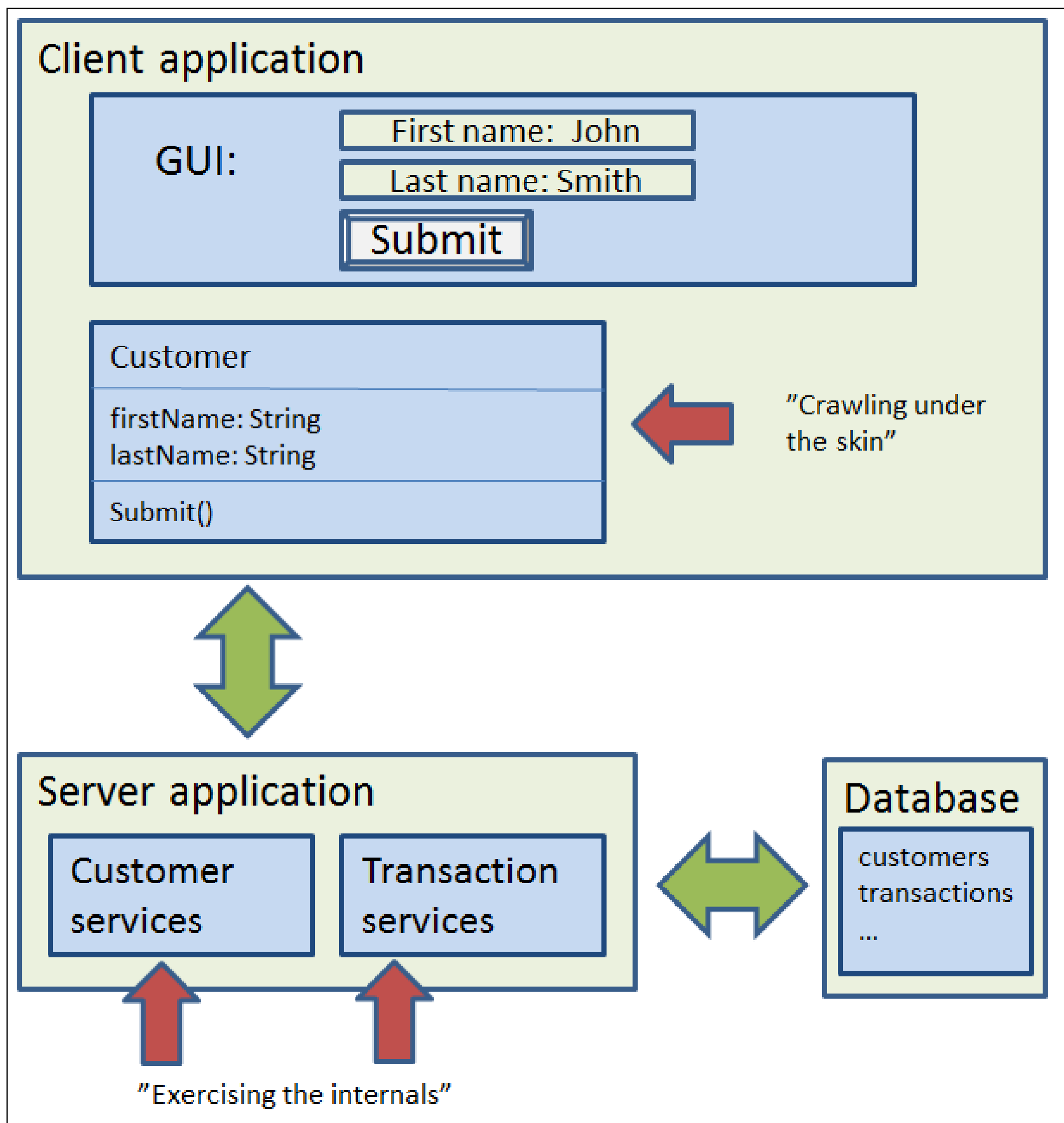


Figure 9.1: 3-tier architecture

9.2 Syntax of FitNesse tests

The following "Hello, World!" example is taken from FitNesse's Two-Minute Example tutorial [42].

eg.Division		
numerator	denominator	quotient?
10	2	5.0
12.6	3	4.2
22	7	~=3.14
9	3	<5
11	2	4<_<6
100	4	33

Figure 9.2: FitNesse Two-Minute Example

The table in Figure 9.2 contains a caption, a header row and several data rows. The caption gives the test a name and at the same time identifies the fixture which is a piece of program code connecting FitNesse to SUT. In a technical sense fixtures are similar to NUnit's test methods (those having "Test" attributes) except fixtures can take parameters.

Header row defines input and output data; columns without the ending ?-sign are input values and those ending with ?-sign are expected outputs. The table above runs six separate calls to the SUT where the first call validates whether the SUT returns 5.0 when numerator and denominator parameters are 10 and 2 respectively.

The table shown in Figure 9.2 is a screen-shot of a web page. FitNesse is implemented as a web application showing all its outputs as HTML pages. FitNesse tests are unfortunately only displayed in a convenient way but cannot be modified directly. The source code of the table shown in Figure 9.2 follows next.

```
|eg.Division                          |
|numerator|denominator|quotient?|
|10            |2                |5.0           |
|12.6          |3                |4.2           |
|22            |7                |~=3.14        |
|9             |3                |<5            |
|11            |2                |4<_<6         |
|100           |4                |33            |
```

Listing 9.1: Source code of Two-Minute Example

The source code looks nice and clean so far. Even a non-technical person shouldn't have any problems creating and modifying such tests. The table above was intentionally made simple to demonstrate the syntax. However, according to experience of the author of this text, the real-world examples inevitably become far more complex and messy. The problem of such hello-worlds is that they, first, hide the cruel reality of complex and hard-to-manage test and, second, they don't expose the true nature of acceptance testing. The Two-Minute Test presented above is a **unit test**, not an acceptance test. It shows the syntax but gives false associations about acceptance testing in general to people only getting familiar with ATDD.

9.3 Specification scenario

ATDD, just like TDD (which was discussed earlier in Section 6.1), is not about testing. It's about **driving** development with acceptance tests. This sentence means that when the testing of a feature gets started, the feature isn't yet existing. Once again the term "testing" is confusing. The concept of ATDD may be hard to explain to testing professionals — they may resist to accept the idea about starting their job before the programmers have done their jobs first. The fact however is that many testing professionals are "domain experts" and testing is not the only kind of job they do. To be able to validate system's behaviour (i.e. to test), they need to know how the system should work and thus they also can elicit software requirements and compose **specifications**. If fact, according to the author's experience, testing is not even the main activity testing professionals often do! So called "testing" may start long before the feature under development might be considered even close to ready. The testers and the developers keep pushing the "ball" to each other until the system gets into the desired state. Such a process looks more like a specification process than a testing.

The following list is a complete list of actions needed to be applied to the system in order to utilize the feature being constructed. Only the last step is about

validating the actual results against the expected. The list is a specification document which is going to become an executable specification (a.k.a. acceptance test or scenario) with help of FitNesse.

1. Create securities
2. Create security issues
3. Add calendar item and process
4. Create customer and portfolio
5. Purchase lots
6. Execute the detachment CA
7. Validate final state

Securities, processes, lots and other items presented in the list above are different artifacts inside the system. For example, the term ″security″ denotes a financial instrument with certain properties like Security ID, Name, Currency etc. The Digia Financial Systems (DIFS) Portfolio Manager's [43] security modification window is presented in Figure 9.3.

In order to get to a point where the code from Chapter 7 (DTCalculator) gets executed, the system needs to be brought into a special state by creating certain artifacts and executing certain processes. Doing this all by hand is possible too, but is very error-prone and takes lots of time.

Unlike software testers who only need to run tests to assure the system works as it is supposed to work, the programmers run tests also to design, explore and optimize the system. The consequence of different goals of testing is that programmers have to run tests for fifty times whereas testers get satisfied with five. To make long story short: routine testing must be automated and that's what testing frameworks like FitNesse are meant for.

9.4 A real example of FitNesse test

The following piece of code shows how two securities get created with a FitNesse ″test″. The first row is an optional comment describing what the following table is going to do. It shows up in the test result as a nice heading. The next row specifies which fixture to be used (RowFixture) and which parameter to pass to fixture's

Figure 9.3: DIFS PM security window

constructor (a string ”InsertTestSecurities”). Next comes the header row; the first column, SecurityID, is an output column (the ?-sign) and all the others are input columns. The table hence creates two securities with different parameters and stores the Security ID into a variable for later use.

```
!3 Create test securities
!|Table:RowFixture|InsertTestSecurities                                          |
|SecurityID?       |BaseName|SecurityCurrency|SecurityType|CountryCode|MarketCode|
|%[sourceSec]=     |CASOURCE|${secCurrency}  |10          |FI         |01        |
|%[srSec]=         |CASR    |${secCurrency}  |15          |FI         |01        |
```

Listing 9.2: Creating test securities

After creation both securities may be opened in a similar window to the one shown in Figure 9.3. They would have something like ”CASOURCE123” and ”CASR234” as their Security IDs (auto-generated), their Currencies would be the contents of FitNesse variable ”${secCurrency}” and their Country code would be

"FI".

Figure 9.4 shows the beginning part of the test after execution.

All other system artifacts are created in a similar way. FitNesse supports nested pages as shown in Figure 9.4. The main page contains references to other pages like DefineVariables, SetupCa etc. Creation of securities and security issues happens inside the SetupCa.

The following listing shows the top level of the DetachmentConfirmation acceptance test.

```
!************> Define variables

!include >DefineVariables

*************!

!************> Setup CA

!include >SetupCa

*************!

!************> Prepare portfolios

!include >PrepareInitialState

*************!

!************> Validate initial state

!include >ValidateInitialState

*************!

!************> Execute CA

!include .SamstockTestSuite.CorporateActions.SharedPages.ExecuteCorporateAction

*************!

!************> Validate final state

!include >ValidateFinalState
!include >ValidateTransactions

*************!
```

Listing 9.3: The top level of a test

Figure 9.4: DetachmentConfirmation test

The top level of the test looks much like the specification list presented earlier. The system gets prepared for the test until the point where the actual execution of the corporate action takes place — the ”Execute CA” block. That block is where the code from Chapter 7 gets actually executed.

Depending on input parameters the SUT (DTCalculator) calculates certain detachment amounts which were also tested by the unit test presented in Chapter 7. These values are checked in the acceptance test too. The following listing shows the *ValidateFinalState* part of the test. Detached amounts are checked in the *AMOUNT* column.

```
!|Table:ListFixture|GetLots|{COM_CODE:%[customerIDA]}                                    |
|TRANS_NR        |PORID   |SECID          |AMOUNT|TRANS_DATE   |VAL_BEGIN    |
|%[sr1]=         |        |%[srSec]       |3     |${issueDate}|${issueDate}|
|%[sr2]=         |        |%[srSec]       |4     |${issueDate}|${issueDate}|
|%[sr3]=         |        |%[srSec]       |3     |${issueDate}|${issueDate}|
|                |        |%[sourceSec]|2     |${issueDate}|${issueDate}|
|                |        |%[sourceSec]|3     |${issueDate}|${issueDate}|
|                |        |%[sourceSec]|4     |${issueDate}|${issueDate}|
|                |        |%[sourceSec]|5     |${issueDate}|${issueDate}|
```

Listing 9.4: ValidateFinalState

So, what is the point in duplicating the tests, one might ask? The point of the acceptance test described in this chapter is not only to test detachment calculations, which is the core of the Detachment CA, but to specify the whole process. While the unit test helped in developing the core algorithm, the acceptance test ensures that all the surrounding artifacts (securities, processes and so on) function as expected and interact with each other. Apart from *AMOUNT* there are many other columns to check. The other column values of the *GetLots* table depend on the actions taken before the CA execution (the *Execute CA* block). An invalid value in any of these fields would break the whole test.

9.5 FitNesse tables

As a reader familiar with FitNesse could have noticed, the tables presented above are not built-in FitNesse tables. These two table types are "Table:RowFixture" and "Table:ListFixture". These are custom tables created by the author of this text and they are not included in FitNesse. The reason why such tables were implemented is because the built-in tables available in FitNesse miss some essential functionalities which will be explained later in this section.

SLIM (Simple List Invocation Method) is the name of a test table style. There are ten different built-in types of SLIM tables, namely the following (FitNesse documentation [44]): Decision, Query, Subset Query, Ordered Query, Script, Table, Import, Comment, Scenario and Library tables. However, only the first two of them (Decision and Query tables) are essential and these two are enough to be able to execute any possible kinds of tests. The rest tables are either slightly different versions of the basic tables (Subset Query, Ordered Query, Script) or they provide some extra functionality which is not directly related to expression of tests.

In Decision Table each table row is an independent instruction having both input and output values or, optionally, either inputs or outputs. The keyword here is

”independent instruction”. The following listing represents an example test table.

```
!|DT:Fixtures.InsertOrder                                |
|transactionNumber?|orderType|symbol|amount|price  |
|                  |Purchase  |Nokia  |1000   |6.45   |
|                  |Sale      |Nokia  |500    |6.94   |
|                  |Purchase  |Apple  |1200   |421.17|
```

Listing 9.5: Decision table

The caption row specifies the table type (DT = Decision Table) and the needed fixture class (Fixtures.InsertOrder). In this case, FitNesse will try to find a class named ”InsertOrder” from a name space ”Fixtures” which in turn resides in an assembly (a DLL-file if it's a .NET program) belonging to the system under test (SUT). The physical path to fixture assemblies is configured with a ”path” setting which is described in FitNesse user guide [45]. Fixtures hence work as an interface between FitNesse and SUT.

The second row specifies test data column names. In this case all but the first column (transactionNumber) are input columns while the first is an output column which is indicated with the ?-sign. The rest three rows are data rows. The whole test inserts three order into the SUT. The first order is a Purchase of 1000 Nokia shares at a price of 6.45. The second order is, consistently, a Sale of 500 Nokia shares at a price of 6.94. The transactionNumber-column is supposed to display a unique transaction number after the insertion (the output).

Decision Table is mainly used for those data manipulations which in ”database world” get handled by INSERT, UPDATE and DELETE SQL statements. Decision Table differers from these SQL statements in one way: the data of each record (row) doesn't only go *in* but can also be fetched back (out) with *the same, single call*.

Query Table instead is not meant to manipulate data but only to fetch it. It is an equivalent of SELECT SQL statement.

```
!|Query:Fixtures.GetTransactions|Purchase              |
|orderType                      |symbol|amount|price  |
|Purchase                       |Nokia  |1000   |6.45   |
|Purchase                       |Apple  |1200   |421.17|
```

Listing 9.6: Query table

The table above is meant to fetch all *purchase* orders. Query table returns a list of records constrained by certain restrictions (orderType here). The Query Table presented above handles data in a similar way to the following SQL statement:

```
SELECT orderType, symbol, amount, price
FROM   table_transaction
WHERE  orderType = 'Purchase'
```

Listing 9.7: Query table SQL

Accordingly, every Decision Table row can be translated into similar INSERT, UPDATE or DELETE statement. Decision Table can also be used for fetching data, but unlike Query Table, it is limited to single result row per query. To sum up, Decision Tables are very good at modifying states of system under test while Query Tables let efficiently verify whether SUT is in a desired state or not.

The reason why the author of this text had to implement custom versions of Decision and Query table (RowFixture and ListFixture respectively) is because built-in versions lack some essential functionalities.

The first and the biggest problem is that Query Table doesn't support variables. The previous example could store transaction numbers into variables instead of simply displaying the values:

```
!|DT:Fixtures.InsertOrder                                |
|transactionNumber?|orderType|symbol|amount|price  |
|$TransNr1=        |Purchase |Nokia |1000  |6.45   |
|$TransNr2=        |Sale     |Nokia |500   |6.94   |
|$TransNr3=        |Purchase |Apple |1200  |421.17|
```

Listing 9.8: FitNesse variables

Variables (TransNr1, TransNr2, TransNr3) can store data values for later processing. Another tests table can now use them for example to update values:

```
!|DT:Fixtures.UpdateOrder|
|transactionNumber|price  |
|$TransNr1        |6.47   |
|$TransNr2        |6.97   |
```

Listing 9.9: Updating values

The problem is that Query Table doesn't support such an essential concept as variables. That is most certainly one of the reasons why FitNesse failed in becoming popular. The author of this work overcame the issue by re-implementing the whole variable processing mechanism.

In the technical sense in means utilizing functionality provided by Table table and manually parsing all the input text coming from FitNesse in a fixture code. When the data have been parsed, it should be searched for variables encoded with some custom marking format such as `%[srSec]`. Next, the text value of `%[srSec]` is replaced with the actual variable value, e.g. `SR172` which is being stored in a

static dictionary having key-value pairs like `%[srSec] = SR172`. Now the data get passed to the system under test. The journey doesn't end here – the most exciting part is still coming. When the data come back from the system under test, each cell with the actual result has to be compared to the expected values initially received and marked accordingly with green, red or grey colour, depending on whether the actual values match the expected, they diverge or are irrelevant (expected value is empty). Finally, variables in the static dictionary should be initialized in case they were marked to be initialized with the = sign (`%[srSec]=`) in the expected data.

Such a modification took quite a lot of time and effort which is obviously not how most people would like to use a new tool.

Another big problem with FitNesse is how Decision Table requires its fixtures to be implemented (FitNesse documentation [46]). Each column of a Decision Table requires its own method in a fixture class. So, if a domain object has 200 fields (which is not uncommon), then the corresponding fixture must have 200 getter-methods for value outputs and 200 setter-methods for inputs, totalling 400 methods. This turned out to be a maintenance burden because fixtures needed to be changed whenever corresponding domain object (e.g. order) got a new field or a field was renamed or removed. This kind of structure doesn't make sense because all these value manipulations can be accomplished in an easy and elegant way with reflection [47]. The author had nothing else to do but to implement the whole Decision Table functionality from scratch which took a great deal of time, again.

The final design consumed only one built-in table: the "Table Table", which is a special table type enabling programmers to implement their own test processing logic independently from FitNesse. Unfortunately, it was the only truly useful built-in FitNesse table.

The real problem with FitNesse, however, is not about lack of useful built-in functionality which can be implemented with "Table" table outside of FitNesse. It's about the code editor which is much harder to replace.

9.6 The Big Problem

The following listing shows a transaction validation table. It ensures the SUT has the needed transactions after the CA execution. The transactions are queried with a COM_CODE parameter which was acquired during test initialization (SetupCa) and saved into a variable (customerIDA).

```
!|Table:ListFixture|GetTransactions|{COM_CODE:%[customerIDA]}    |
|TRANS_NR           |PREV_NR        |ACT_NR          |TRANS_CODE|PORID|
|                   |0              |                |11        |     |
|                   |0              |                |11        |     |
|                   |0              |                |11        |     |
|                   |0              |                |11        |     |
|%[sumTransNr]=     |0              |%[sumTransNr]   |95        |     |
|%[sumTransNr2]=    |0              |%[sumTransNr]   |95        |     |
|%[43TransNr1]=     |               |%[sumTransNr]   |43        |     |
|%[43TransNr2]=     |               |%[sumTransNr]   |43        |     |
|%[43TransNr3]=     |               |%[sumTransNr]   |43        |     |
|%[42TransNr1]=     |%[source1]     |%[sumTransNr]   |42        |     |
|%[42TransNr2]=     |%[source2]     |%[sumTransNr]   |42        |     |
|%[42TransNr3]=     |%[source3]     |%[sumTransNr]   |42        |     |
|%[42TransNr4]=     |%[source4]     |%[sumTransNr]   |42        |     |
```

Listing 9.10: Transaction validation

The listing presented here contains only five columns in order to fit into width limits of a static medium such as this document. The real test contains over thirty columns while the SUT has over 200 of them. This makes test management quite a complicated task. For example, removing a column using FitNesse's code editor requires selecting column contents and removing it row by row. This is a very unproductive way of working especially considering how easy it can be done in common spreadsheet applications. FitNesse's code editor is actually its biggest weakness at least when the system under test has a lot of columns.

Poor code editor is not only annoying to work with but it also has a bad impact on overall test readability. FitNesse tables are often very similar to each other and only data values differ, so it's reasonable to use table templates. However, templates tend to grow over time and new columns appear. All the columns are yet not needed in every place but they stuck in tests because they are so hard to remove. The whole test suite ends up containing huge tables with only a few columns being actually checked making tests look very unclean and hard to navigate. A testing professional seeing such a mess for the first time won't like it for sure. The first impression is the most important, they say, and FitNesse's code editor is not the best thing to introduce to ATDD newcomers. FitNesse definitely needs a better user interface.

The problem of a bad table editor seems to be highly underestimated. There had been several topics about the issue on FitNesse discussion groups, but these topics failed to generate any significant conversation. Almost like the creators and maintainers of FitNesse never had a chance to actually use it themselves to develop large-scale software projects with high data amounts.

A logical suggestion is to use an external tool (e.g. Excel) to modify test data. Some text editors (e.g. Notepad++) support selecting a column by pressing and

holding the Alt button and selecting the needed text area with a mouse. Spreadsheet tools are however much handier in handling tabular data.

FitNesse has a feature which translates a test page into a format suitable for copy-pasting into Microsoft Excel or other spreadsheet applications. The work flow is the following: first, the test page is opened in FitNesse's own code editor. Next, all the text is made copy-pastable by pressing a button "FitNesse to spreadsheet". Then, the text can be selected and copied into Excel where the needed modifications are made. The process then continues in reverse order — the needed text is copied and pasted from Excel to FitNesse and translated back by clicking "Spreadsheet to FitNesse".

According to the author's experience, such a process doesn't work simply because there are too many steps to be done in order to perform such a simple action as removing or adding a new column. It's also too easy to get confused and make changes to both text versions by accident. The user will then have to find out which changes are needed and which not. Or worse, one of the changes will be lost by overwriting with the other.

Another option could be to make spreadsheets (or text editors) the primary data source and generate the actual test code with macros, for example. However, such an approach has been already tried earlier with Fit. It lacks the possibility of efficient test organizing and refactoring, which was the main reason to build FitNesse in the first place.

To sum up, FitNesse has a lot of good features but involves too much accidental complexity (see Chapter 2) to be efficiently and successfully used in a large-scale project. The following chapter is going to sketch a "better tool", a tool for specification driven development which takes the best features from FitNesse and adds the missing ones.

10 The Missing Link

The previous chapter introduced FitNesse working in practice. Unfortunately, FitNesse has some critical drawbacks making it too hard to use in certain environments. According to the author's personal experience, FitNesse requires a great deal of personal commitment for successful adoption. Also the fact that FitNesse failed in becoming truly popular speaks for itself. Its code editor requires too much work to maintain tests in a good shape. While a motivated programmer could deal with a clumsy text editor, it seems to require too much effort from domain experts accustomed to convenient spreadsheet applications. This is an unacceptable attribute for a communication tool, a *link* between programmers and domain experts which FitNesse is supposed to be.

10.1 Existing tools are just not good enough

Meanwhile, FitNesse has at least two "killer features". First, FitNesse is a test organizer. It provides the ability to structure, execute, modify, search and refactor executable specifications. These are probably the main goals behind FitNesse's design and which are missing from its predecessor, Fit. These goals were actually quite well met.

The second essential aspect is tabular communication model. FitNesse (and Fit) communicate with the system under test in a different way than BDD tools do. The latter try to produce as clear and readable specification documents as possible. Tabular-styled FitNesse in turn handles data in a database-like form. FitNesse tables may be compared to SQL statements; some tables change system's state (insert, update, delete) while some are query-tables (select).

The tabular form is much less human-friendly than the Given-When-Then form but it suits data-rich applications like financial software described in previous chapters much better. BDD tools emphasize hiding implementation details inside test code and expose its users to only a truly necessary logic (i.e. behaviour). This approach doesn't suit data-rich applications, because their behaviour is not the only thing needed to be tested; a great deal of data needs to be validated additionally to behaviour. Hiding such data from domain experts in program code doesn't serve the purpose of an ATDD communication tool.

Figure 10.1 is a screen-shot of another popular automatic testing tool – SoapUI [48]. The upper part of the screen-shot shows a list of test results. Each item (CustomTransaction) corresponds to a row in FitNesse table. Each item has multiple properties (Amount, ClearingDate etc.) which correspond to columns in FitNesse. The row selected in the screen-shot contains an error displayed in the lower part — the expected value (53789645) doesn't match the actual (1204088601).

SoapUI doesn't provide an easy way to locate the failing point in a test data. The assertion shown in Figure 10.2 needs to be investigated. It contains an XPath link to the failing point meaning that a user is forced to manually search for the failing value by counting items according to indexes saved in the assertion. In this case the failing item is the second in the list (CustomTransaction[2]) but it could equally be a hundred-thirty-seventh making it very hard and error-prone to find.

Likewise FitNesse, also SoapUI seems to miss user-friendliness very much. Both tools provide a great deal of functionality but they both forget about the user. Especially less technical people will most certainly refuse to use them because of their low usability. Once again, this is not acceptable due to high communicational value of an ATDD tool.

There seems to be a big hole in ATDD/BDD open-source tool range. The author of this work failed in finding a tool providing such essential properties as sufficient support for process/behaviour modelling, effective tabular specifications and sufficient usability to employ the tool among both programmers and domain experts. The following section tries to sketch what kind of features a decent ATDD tool should have.

10.2 Sketching a better solution

First of all, the Tool must be easy and enjoyable to use. The results of a "Survey of Agile Tool Usage and Needs" [49] indicate that ease of use is clearly the most valued aspect of agile tools. Tabular specification data must be easy to manage — adding and removing rows and columns should be trivial which is not the case in FitNesse. Structural specification management must be supported, i.e. organizing, searching, refactoring and other good features found in FitNesse. Finally, the exact location of a failing point of a test must be accessible with a single mouse click (missing from both FitNesse and SoapUI).

These are relatively small improvements to FitNesse yet having a great usability improvement potential. However, to create a truly effective tool, something must be done to tabular data. Humans are just not capable of handling big amounts of tab-

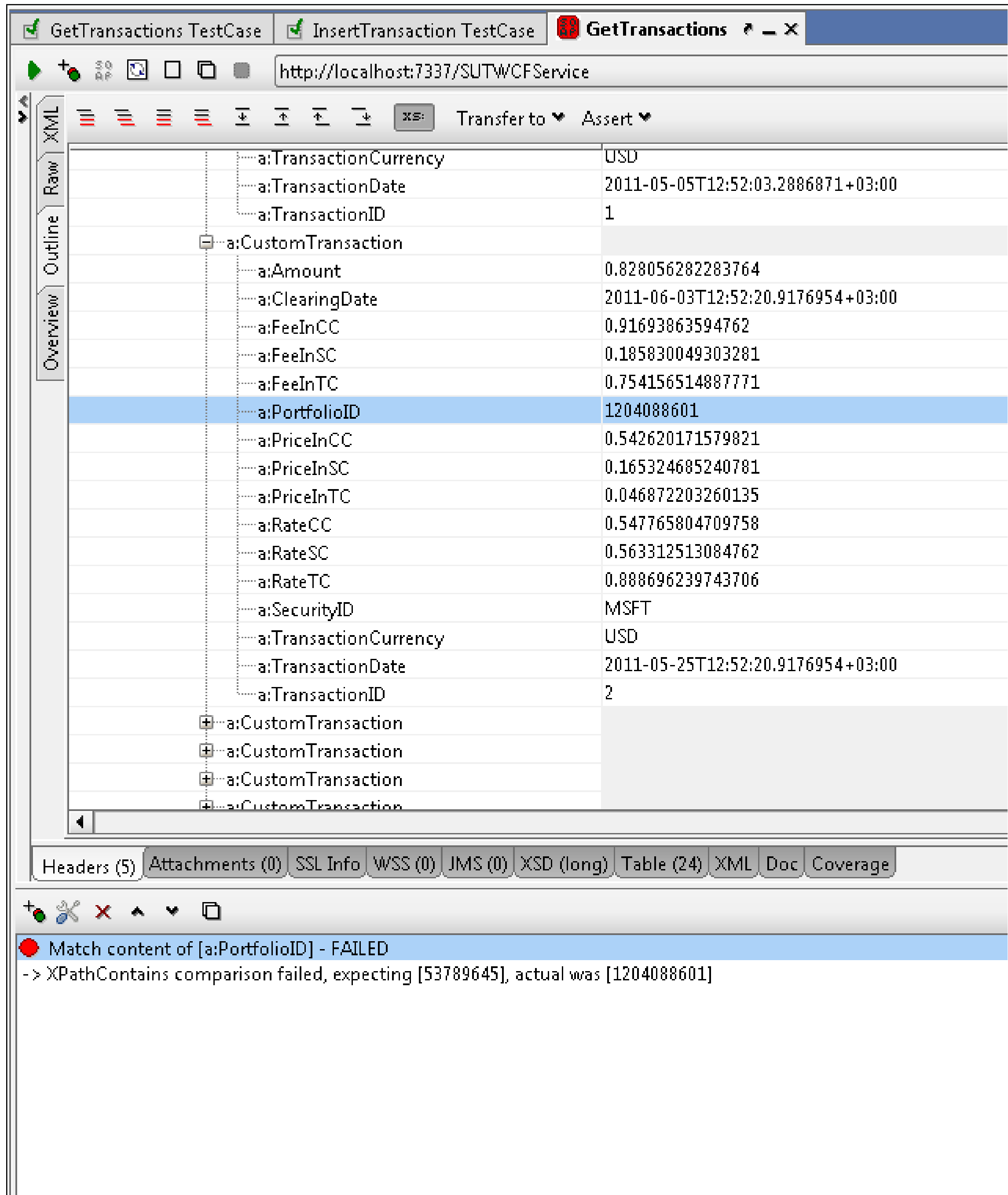


Figure 10.1: SoapUI test result

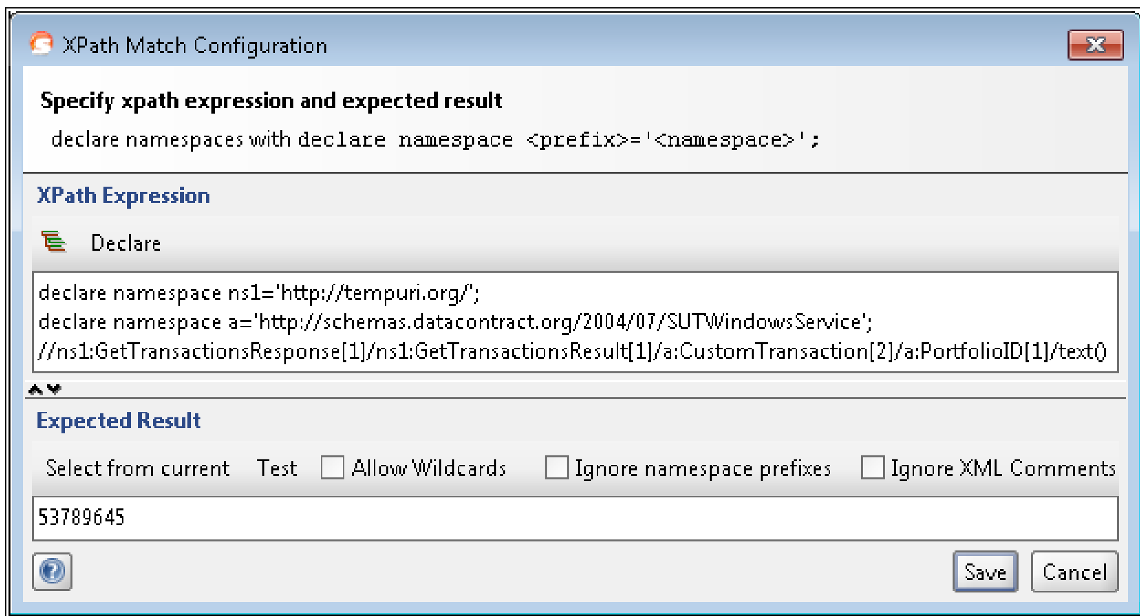


Figure 10.2: SoapUI assertion

ular information effectively and hence tabular data needs to be separated from behaviour specifications but remain easily available when needed. That would make the tool more BDD-like.

Every person who had been somehow involved in construction of software is most likely familiar with visual process models like sequence, activity, use case or other kinds of diagrams. Human brains are well trained to process visual information, so why not hide all the specification tables behind nice pictures of system states and state transitions?

Figure 10.3 shows an imaginary executable specification expressed as a state diagram. Instead of running tables one after another, the execution process is visually modelled by a human-readable state transition chart. Visual state transition diagrams provide some extremely beneficial features.

First, more than one separate tests can be effectively included in a single picture making code re-use more easy and test dependencies obvious. Figure 10.3 shows two tests (”Insert valid transactions” and ”Insert invalid transactions”) which both share a common initialization (”Initialize SUT”).

Second, each test can be divided into as many sub-tests as needed. For example, the state transition ”Initialize SUT” consists of three other state transitions which insert securities, a customer and portfolios respectively.

Finally, the actual tabular data resides on the leaf nodes of a diagram. Figure 10.4 shows the contents of the ”Execute confirmation 1” state transition. It has three

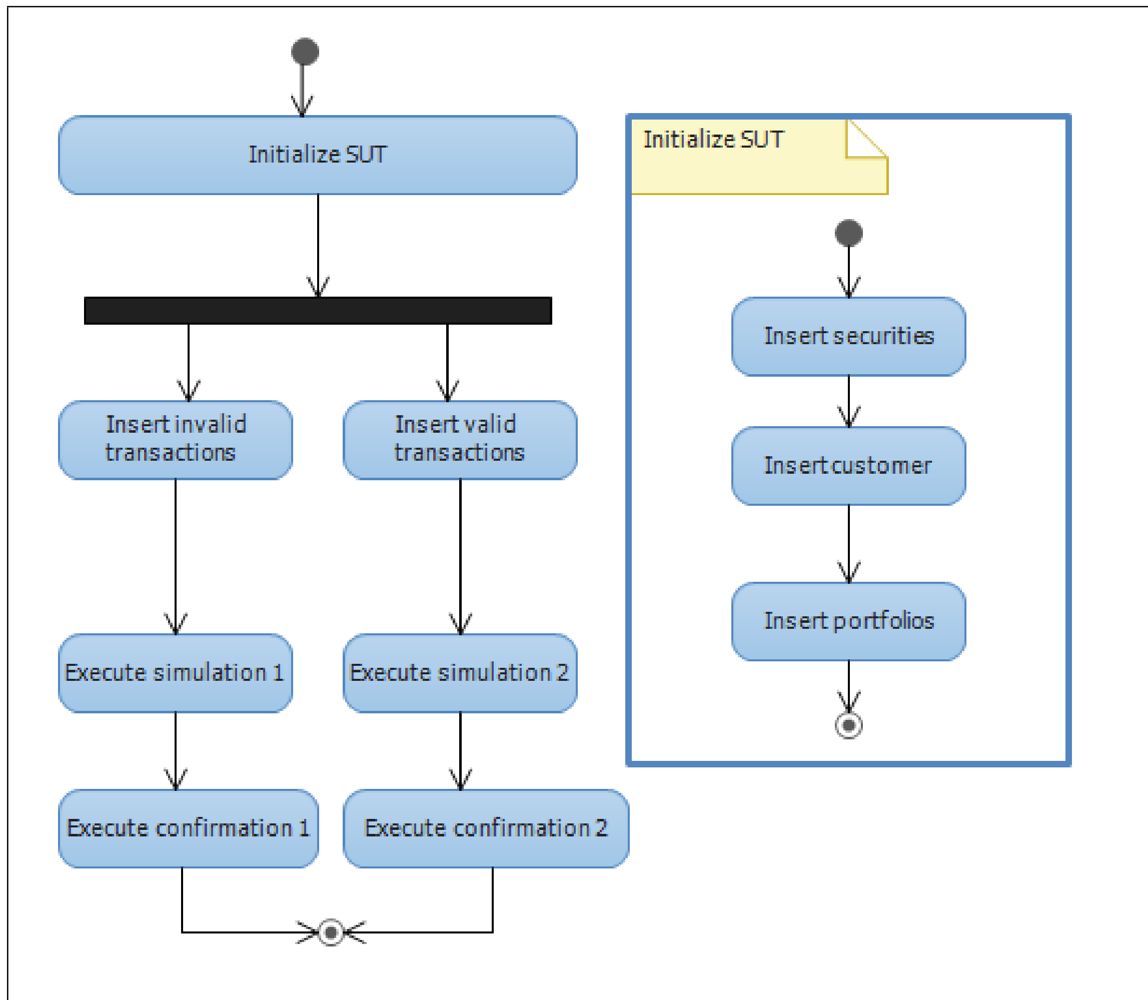


Figure 10.3: Executable specification diagram

tables; the first launches the actual state transition and the other two tables validate the new state. The cells marked with grey color contain variables and those marked with blue color (dates) – are constants.

Execute confirmation 1

	A	B	C	D	E	F
1	**CActions.Execute**					
2	ProcessID	ProcessPhase				
3	procID	Confirmation				
4						
5	**TransactionEngine.GetLots**					
6	COM_CODE					
7	custA					
8	TRANS_NI	PORID	SECID	AMOUNT	TRANS_DT	VAL_BEGIN
9	sr1	p1	srSec	3	4.3.2011	4.3.2011
10	sr2	p1	srSec	4	4.3.2011	4.3.2011
11						
12	**TransactionEngine.GetTransactions**					
13	COM_CODE					
14	custA					
15	TRANS_NI	PREV_NR	ACT_NR	TR_CODE	PORID	
16		0		11	p1	
17	tr1=	0	tr1	95	p1	
18	tr2=	0	tr2	95	p1	
19	tr3=		tr3	43	p1	

Figure 10.4: Executable specification tables

This chapter expressed the personal opinion of this text's author about what a decent ATDD tool should look like. The opinion is mainly based on working experience with FitNesse and developing data-rich applications. So far such a tool seems to be missing from open-source tool range.

However, its very likely there are similar commercial tools already existing. They are quite hard to find, because they usually include a vast amount of other features additionally to the needed ones. Also, not all commercial tools provide an evaluation version to try the tool and even if they do, evaluation versions are often hard to

acquire due to mandatory registrations and other arrangements. Finally, they often cost fortunes.

The idea about using visual diagrams as test logic containers came from the author's colleague Tuukka Turto and it seems there is already a similar implementation existing. One tool enabling acceptance testing through activity diagrams is Microsoft Visual Studio 2010 Ultimate. Figure 10.3 was created using Visual Studio's Activity Diagram editor. Visual Studio enables creation of such diagrams and, additionally, attachment of executable tests to each diagram item. However, such functionality requires Team Foundation Server 2010 which was not available to the author of this work and hence the actual process of test-driving an application was not carried out. Additionally, it remains unclear whether Visual Studio supports tabular tests or not and how well the implementation serves the purpose if tabular tests are supported.

11 Available literature and case studies

This thesis is not a genuine scientific piece of work because conclusions made in previous chapters were mainly based on the author's personal experience and available information sources which are often non-scientific. Many sources provided in this work may not be considered academic material as they are not results of case studies or other research projects conforming to common scientific research norms. The knowledge gathered throughout this work mainly came from books, articles published in blogs and even public forum discussions.

This work is not based upon a well-known research problem all aspects of which have been studied for decades. Libraries don't burst with high-quality academic literature criticizing and defending the subject from different perspectives. In fact, as of September 2011, *no single article* could be found in IEEE Xplore digital library with the keyword ATDD. ATDD doesn't even have its own page in Wikipedia. Other search keywords such as "automated acceptance testing" (AAT) work slightly better, but still, the amount of *versatile* literature is very small. The reason for poor availability of scientific articles discussing ATDD probably lies in relative immaturity of the subject and in lack of successful case studies which, in turn, prevents software producers from adopting the technique. It's a vicious circle.

The current chapter might not had existed unless the author stumbled upon an excellent article presented at AGILE2011 conference by Børge Haugset and Geir K. Hanssen — "The home ground of Automated Acceptance Testing: Mature use of FitNesse" [50].

The article references two other articles written earlier by the same authors: "Automated Acceptance Testing Using Fit", 2009, [51] and "Automated Acceptance Testing: a Literature Review and an Industrial Case Study", 2008 [52].

These three articles, as well as other sources cited by them, are going to be further reviewed throughout this chapter.

The study described in the first article [50] is based on in-depth interviews with four software consultants who, citing the article, "think they successfully use AAT" (Automated Acceptance Testing). The research question being answered in the study is:

> "How and why is FitNesse used in a successful and mature software development project?"

One of the main outcomes of the study is that FitNesse *can* be successfully used for development of complex software. It can significantly improve quality of software by making it much easier to develop and maintain. FitNesse tests build up a safety harness which makes program code much less fragile and enables developers to make changes and improvements without fear of breaking something. Executable specifications make system under development much easier to understand for developers at the same time improving communication among developers and making program code much easier to share. The test harness serves as a good system documentation for developers.

On the other, negative side, there was *no single case* in which customer participated in specifying a system with executable acceptance tests. Customers refuse to write tests in tabular format and prefer conventional documentation over executable specifications even if they promised great benefits.

The current work haven't been influenced in any way by the articles mentioned above. The author of this work discovered these articles only after having finished all the earlier chapters. Nevertheless, the findings described in the articles fully agree with conclusions emerged throughout the previous chapters of this work. This fact suggests that these conclusions can be generalized at least at some level even if none of the studies forming the base for this discussion report quantitative data documenting the net benefit of AAT [51][p.7].

The following sections are going to walk through the main conclusions Hanssen and Haugset made in their three articles. These conclusions are based on literature reviews, industrial case studies and interviews with developers practicing AAT.

11.1 Common knowledge about AAT

AAT is much like unit testing but on a higher, understandable to users (i.e. non-programmers) level [51][p.1]. Acceptance tests (a.k.a. executable specifications) are meant to repeatedly and automatically test software at a business level [50][p.97]. AAT is meant to solve three main problems: communication, agility and balance [53].

Communication here means that ordinary written requirements can be ambiguous, prone to misunderstanding and can lead to wrong design. Tabular tests tend to improve communication about software requirements through realistic executable usage scenarios.

Agility is supported by giving the possibility of frequent changes due to safety harness providing instant feedback to developers whenever the system gets broken.

Balance is a bit unclear concept which seems to overlap with the Agility part. It is explained as follows [53][p.301]:

> ”Balance: Spending less time on gaining balance by reducing the number and severity of problems, catching them early and making sure they don’t return. As software systems age, they’re inclined to get more and more difficult to change. Effort is needed, through refactorings, to keep systems in balance. To do this effectively, however, fast, automated tests are needed to help ensure that changes don’t introduce bugs.”.

The latest article (2011) [50], however, divides main improvement targets (goals) of AAT into two parts: Specification and Verification. This is exactly the same division the author of this work came up with in Chapter 5 (The main goals of ATDD), although *specification* and *verification* are precise terms revealing the true essence. They’re meaning is explained in terms of knowledge flow and defined as follows [50][p.102]:

> ”Step 1: (Specification) Knowledge of the business and problem domain is transferred from the customer to the developers and,
>
> Step 2: (Verification) Knowledge about how the software meets the requirements (AAT test outcome) is communicated back to the developers”

The following sections are going to review how well the goals presented above have been achieved with the means of AAT in recent case studies.

11.2 Customers don’t write tests

The most important observation confirmed by several sources is that customers neither write automatic acceptance tests nor evaluate the results [54], [55], [51][p.5], [52][p.34]. This fact is also considered to be the biggest problem of AAT [51][p.8].

There are no clear explanations available about which are the exact reasons behind customers’ refusal of executable specifications. According to experience of this work’s author, customers (i.e. domain experts) tend to avoid answering questions about why they don’t want to write acceptance tests. One reason could be that tools are not good enough, like it’s suggested in Chapter 10. On the other hand, according to sources listed above, customers also refuse to write specifications with Fit which doesn’t have problems with editing tabular data like FitNesse does.

Another reason for refusing to write test may lie in business analysts not being interested in writing more than plain text and wanting to leave it to QA analysts to code the tests [55]. One customer explained that it was easy to skip writing tests because they are complex beasts and "... the thought of diving into that complexity, just when we thought we were done, was unpleasant" [52][p.31].

Customers prefer to express requirements in traditional way and in meetings [52][p.34]. That way they don't only do things the way they are accustomed to but they also keep *control* of their schedule. Business people, like the term suggests, are busy people. They only have certain amounts of time available for each task. They simply can't afford spending as much time as required by executable specifications which elicit much deeper and comprehensive insight of the problem, not only the nice "big picture". Nevertheless, customers don't have other choice but to fully specify the system, at the latest when the system is in production and they have to file bug reports. By that time the price of fixing the "bug" may have grown by a couple of orders of magnitude.

Most customers don't write tests but that doesn't mean they can't read them indirectly. At least one study reports success in employing Excel as a mediator between executable acceptance tests and customers [50][p.100-102]. In that project FitNesse tables were written by developers, then the needed data was transfered to Excel and sent to customer for review. The customer checked the data, filled in the needed values and sent the excels back to developers who then fixed the tests according to excels.

Such arrangement is not ideal because it results in extensive use of added document, which is actually very typical of AAT projects [50][p.98], [56]. Moreover, because executable specifications as such were not used in communication with customer, they haven't become a documentation suitable for customers' needs [51][p.5]. Still, using a mediator achieves one big goal — specifications are being communicated in form of usage examples and not as free prose. As the conclusions-part of [50][p.104] states, FitNesse doesn't introduce just the tool, but also the way of thinking.

To sum up, it seems that giving customers the opportunity to write tests is not enough: "The difficulty was not the practice (of writing tests) itself, but the discipline of doing it" [52][p.31]. There has to be some process or a work model to obey [52][p.36]. Storytest-Driven Development is suggested as one such development model [52][p.36].

Another viable solution is to shift the role of customer to someone who would be more willing to follow the rules of development team without being a programmer.

In Scrum the best candidate for playing customer's role and writing acceptance tests is product owner.

11.3 Other common issues

Unfortunately, AAT is definitely not a silver bullet. As it was mentioned earlier, customers don't write tests which defies the main point of AAT. However, there are some other issues too.

Although tabular format itself is very easy to learn and understand even for non-technical people ([50][p.98], [56]), experience with AAT and process organization have a crucial role considering success of utilization of the technique [57]. Lack of experience doesn't necessary mean the project is going to fail but it results in high adoption costs [51][p.5].

For example, tests tend to grow large and become unreadable [51][p.5]. It is a severe problem because too complex tests waste lots of time on maintenance threatening adoption of the technique ([51][p.6], [50][p.97]). However, this is mainly a problem of unexperienced developers [51][p.6], because experienced ones know how important is it to keep tests readable and they also know how to achieve it. In case of FitNesse it turned out to be beneficial to divide tests into many separate tests pages so that each table goes to its own page. That way tests remain readable even if tables had dozens of columns. Unexperienced developers also tend to put too much effort into expressing correct executable specifications at the expense of producing only appropriate tests which hinders communication [55].

Another problem making acceptance tests hard to maintain is lack of support for refactoring [58]. This was one of the most evident findings of empirical part of this work (see Chapter 10). Messy tests, in turn, are time-consuming and unpleasant to work with and hence writing tests has at times been given lower priority compared to programming which is also confirmed by other studies [59]. Writing acceptance tests is easy to skip also because they can take a lot more time compared to unit tests [60].

Apart from becoming complex, acceptance test suites tend to become hard to organize. Both Fit and FitNesse rely on folder structure for organization which has its drawbacks. In one project [55] tests were started to be kept in different directories according to their life cycle status. This caused problems when people did not move documents properly. Another major problem with folder-based organization is that maximum path length in Microsoft Windows (at least up until Windows 7) is around 256 characters which prevents creation of new tests when the total path

length exceeds the limit. At least Selenium [61] and Robot Framework [62] support tags which have proved to be an efficient way of organizing tests [63].

Organizing test becomes overly tricky also because of too high test coverage. Nearly full coverage is a desirable state for unit testing but such a technique doesn't seem to work for AAT. Several case studies (including this work) report the same phenomena: in the beginning developers tend to define tests for most issues, but over time they start choosing only the most complex ones [51][p.4]. Choosing only the most complex requirements greatly reduces amount of tests making test harness much easier to refactor, while at the same time, dropping less complex scenarios doesn't decrease too much value ([57], [64], [65]). Some requirements are better to skip not because they were too simple but because they are generally hard to test and automate. Too abstract requirements such as "good usability", goals and visions should be naturally left to human testers ([50][p.101], [66]). A good rule of thumb is: "if a test doesn't reduce uncertainty, then skip it" [50][p.99].

Relatively small availability of academic articles evaluating AAT is also a problem. AAT is not researched very much compared to unit testing which is the most known, practiced and researched approach to agile testing [51][p.1]. General information is too hard to find and there is too little of it available ([50][p.97,99],[51][p.2]). Some particular subjects like FitNesse benefits and how to properly use it, are especially narrowly studied [50][p.98]. Also there is no industrial experience publicly available about what it takes for non-IT professional to specify requirements as acceptance tests [51][p.7]. Moreover, none of the studies forming the base for this discussion (not this work neither any other study mentioned here) report quantitative data documenting the net benefit of AAT [51][p.7]. Finally, almost all the case studies referenced in this chapter were conducted by student groups during university courses which greatly reduces their practical value by leaving out of context such essential factors as resistance to change, deadline pressure, vast amounts of legacy code, complex hierarchical relations between project members and other attributes of a real life software project.

Unfortunately, the downtrend seems to continue: only a few empirical studies have been published since literature review [52] issued by B. Haugset and G. K. Hanssen in 2008 [50][p.98].

11.4 Benefits for development process and adoption costs

Even though AAT has a bunch of drawbacks and doesn't seem to work for customers, which is a good subject for further research, it clearly does work for devel-

opers.

In all case studies described in articles [51][p.6], [67], [52][p.35], all developers responded positively whether they would like to continue practicing AAT. There are many reasons for such attitude.

First of all, test harness becomes a reliable safety net over time. It makes programming much less of a tightrope walking and much more of a jigsaw puzzle: several combinations can be tried before choosing the optimal one. Mistakes are allowed (most will be quickly caught by test harness) and even encouraged (people learn best from their own mistakes).

Further, acceptance tests make changes safer to do and fewer errors remain unnoticed until manual testing phase [52][p.6]. AAT also makes developers able to see more integration issues [52][p.7]. The latter seems to be a direct consequence of focusing on playing with "puzzle pieces" (components of acceptance tests) instead of trying to keep balance on the rope (trying to figure out which part of code should be changed to get expected outcome without breaking the code elsewhere).

Tightrope walking (i.e. programming complex solutions without covering the code with automatic test) is an art, but that art is for individuals. Even clear design of a complex program will seem complex for other developers besides the creator. One of the most evident findings of these studies [51][p.7] is that developers find it more convenient to share code with other developers which is extremely beneficial phenomenon considering the fact of close cooperation being the key for success [50][p.102]. Executable specifications both give the freedom to experiment with others' code without being afraid of breaking something and also they document program code by storing information (input/output values) about how each feature is supposed to be used.

AAT supports agile development in general by making changes easier to do [51][p.7] and providing *frequent* feedback and validation of solution against requirements [50][p.101,103]. Executable specifications also improve communication by contributing to visibility and process control of SCRUM [51][p.5]. They show graphically which modules are finished so that even a non-programmer can easily get fast and precise feedback about development status [52][p.34].

It is important to point out that communicational and documentational values get reduced when tests grow too large and messy [52][p.34]. Adoption of AAT requires significant investments, especially if there is lack of experience [51][p.7]. Maintenance is the biggest cost of adoption of AAT [51][p.6], but, according to experience of the author of this work, amount of needed maintenance greatly decreases as experience with AAT grows and certain test structure patterns occur. At the end

of the software project described in previous chapters the tests look completely different compared to the beginning. Even though the tests written in the end covered more functionality, they became much cleaner, consistent and easier to refactor. Introduction of new types of test also required minutes of development time in the end of the project compared to days in the beginning.

11.5 Test-driven requirement elicitation

AAT improves quality of both requirements and the process of requirement elicitation by employing a test-oriented description approach [64]. Executable specifications provide a working way of eliciting common understanding about requirements and focusing attention away of loose talks to gain a *result* in communicating requirements [68], [57].

The actual process of creating executable specifications improves understanding of requirements [64]. Apart from keeping conversation focused, executable specifications have been attributed to help discovering a lot of missing pieces and inconsistencies in a story [59]. Also, compared to code-centric unit tests, clear separation of tests and code (an attribute of executable specifications) results in a better overview of what the system is intended to do instead of *how* it does it in technical sense [51][p.8].

Value of conversation around tables is very high when the domain easily fits into table format [50][p.101] which is the case at least for financial and logistics software sectors [50][p.103].

Another great benefit of executable specifications is that they actually can be used for development [50][p.102]. Unlike conventional documents which never tell the whole truth because natural spoken language is always subjective and prone to ambiguity, executable specifications define exactly which input values should result in which outputs. They document the system in objective way preventing misunderstandings. Being executable also enables developers to update specifications as development goes by which, in turn, helps in keeping track of rapidly changing requirements [50][p.102].

Test harness doesn't only protect developers from breaking the system but also from too active customers who want it all. The tests make conflicting requirements more concrete by visualizing them: only one of the two conflicting tests can be green at one time, not both. If customers disagree, they have to clearly state which values should be displayed on which tests or to abandon one of them if it's impossible. In case of conventional documents discussion may go off track because the counter-

parties don’t fully understand each other. Sometimes, however, people simply want to be right, no matter what the problem is. That might be one more reason why they don’t write executable specifications.

12 Summary

This work discusses test-driven development in general and concentrates on Acceptance Test-Driven Development. The main purpose of this work was to empirically evaluate adoption of ATDD in a complex legacy software project. The project team didn't have any prior experience about automated testing neither program code had any automatic tests before this work.

This work can be roughly divided into three parts: introduction, empirical and literature review.

The introduction part includes the background and general knowledge about techniques being discussed throughout this paper. Chapter 2 starts with the review of Frederick Brooks' essay "No Silver Bullet: Essence and Accidents of Software Engineering" which, in opinion of the author of this work, describes the most important problem of software development in general, the root of all evil.

The essay states that all software complexity can be divided into two types: essential and accidental. The former relates to software being essentially complex which means such complexity can't be reduced without sacrificing the software requirements. However, essential complexity can be made easier to manage by adopting good engineering practices like ATDD. In turn, the origins of accidental complexity come from poorly designed and implemented tools and practices resulting in inefficient working process when significant part of human work consists of routine tasks not related to the actual problem. As examples of such kind of complexity Brooks provides high-level programming languages and object-oriented programming which led to enormous productivity gains compared to assembly languages. The biggest problem of ATDD also relates to accidental complexity which is one of the *main conclusions* of this work — existing tools are still too hard to use to fulfill their purpose.

Chapter 2 also reviews convenient software development models (Waterfall and V-model) and discusses the common issues of plan-driven development, such as postponing the testing phase until the end of a project when there isn't enough time to test properly and letting documentation become outdated.

Chapter 3 discusses the common reasons behind failures of large-scale software projects. The chapter lists the most obvious reasons but also presents the article "Why Big Software Projects Fail: The 12 Key Questions" written by Watts S. Humphrey,

the "Father of software quality". The article concludes to such topics as bad management visibility, inability to provide and follow precise work estimates, autocratic management, lack of trust and commitment and inability to follow precisely planned development process. Internally, the article leads to introduction of a software development model TSP developed by the author, but in context of this work it is a preface for the following chapter which tells about Agile software development.

Chapter 4 introduces Agile development which tries to overcome issues of plan-driven models, for example, by dividing a big problem into many small tasks and including all the phases of plan-driven processes in each development task. The chapter introduces Agile Manifesto and Scrum, the most popular agile software development model. It also serves as an introduction to the main topics of this work – test-driven development and ATDD.

Chapters 5 and 6 present the main techniques discussed in this work – ATDD and TDD. They tell about the main goals and pitfalls behind these techniques, describe the development processes and address the common misconceptions related to apparent similarity of these two techniques. In brief, ATDD stands for efficient communication between team members, up-to-date executable documentation and possibility of safe software maintenance and refactoring. The main artifacts of ATDD are acceptance tests or, a better term, executable specifications which provide a communication medium to be shared inside a development team between programmers, testers and domain experts, and additionally, between team and stakeholders outside the team (e.g. customers). Executable nature of specifications requires to include all the information needed to implement a feature, not only "the big picture". Such approach helps to elicit hidden requirements and include them early enough when the cost of change is still low.

Unlike conventional static documentation, executable specifications have much better chance of not becoming outdated and eventually abandoned, because they constantly track the state of software and immediately signal whenever the framework of executable specifications disagrees with the system under test, i.e. when an acceptance test fails. When such thing happens, a programmer can respond quickly by either fixing the code to make the test pass or fixing the test if it fails because of a change in domain logic.

The third big goal of ATDD is building a comprehensive safety network of acceptance tests to be able to efficiently maintain and refactor software code. This part is where ATDD overlaps with TDD the most. The main goal of TDD is also to keep program code clean with help of a safety network formed by unit tests.

ATDD promises many great benefits but has lots of drawbacks too. First of all, it

requires significant investments especially if there isn't enough experience of practicing the technique. Discovering best practices of writing acceptance tests can take some time. Without proper testing patterns tests easily become messy and hard to follow. They have to be constantly maintained just like normal program code or they will eventually become useless and the whole technique will be abandoned.

Chapter 6 introduces TDD by uncovering the most common misconception about all test-driven techniques — these techniques are not *testing* techniques. The chapter proceeds with TDD process description, its goals and common issues and finishes with describing the relationship between TDD and ATDD, that is what kind of features each technique should be used for and by whom.

The difference between ATDD and TDD turned out to be very hard to explain and thus they are further evaluated through concrete examples.

Chapter 7 presents an example of a proper unit test, i.e. a test not depending on anything else but the actual code unit being tested. The helps to develop a calculation algorithm which is implemented in a single class (a code unit). The chapter further discusses the main goal of TDD (clean code) and limitations of unit tests.

Chapter 8 describes Behaviour-Driven Development, a transitional technique between TDD and ATDD. The goal of TDD is to move away from code-centric unit testing toward area of domain logic. BDD emphasizes the importance of logic behaviour and simplicity of test representation. The chapter proceeds with description of the main problem of BDD – it doesn't suite any kind of software. BDD emphasizes behaviour and abstracts away test data to make tests look simple and readable. However, behaviour is not the only property making software complex. For example, financial software has high amount of important data needed to be also accessible by users of testing frameworks. The principle of abstracting such data away doesn't serve the purpose of a tool for improving communication between stakeholders.

Chapter 9 continues the discussion about data-rich applications and the need to access test data additionally to behaviour. It introduces FitNesse, the tool used in this project, and presents a real life example of acceptance test which prepares the system under test for the test, executes the actual state transition and evaluates the outcomes. The chapter also discusses different interfaces which are the entry points for testing frameworks accessing the system under test. In the same context the inappropriateness of GUI testing with respect to ATDD is discussed.

The experience of applying ATDD with FitNesse revealed some major problems with the tool, namely its code editor is not suitable for editing high amounts of tabular data. Chapter 9 describes the issue in detail in its last section while Chapter 10

draws a sketch about the features missed by the author of this text during working with FitNesse. The chapter also introduces SoapUI, a tool for testing applications through web service interfaces.

The last chapter before Summary, Chapter 11, was born almost by accident when the author stumbled upon a very good article written by Børge Haugset and Geir K. Hanssen — "The home ground of Automated Acceptance Testing: Mature use of FitNesse". The article discusses how FitNesse can be successfully employed for test-driving a software project. The article references two other articles by the same authors as well as several other studies. The articles reviewed in the chapter strongly support the conclusions made throughout this work even though they did not influence this work in any way – the author found these articles only after finishing all the previous chapters. Chapter 11 brings at least some amount of scientifically competent material into this work which misses such material a lot. It also supports claims about the lack of versatile scientific information discussing ATDD.

This work has been a great learning experience for its author. The technique had to be learned "the hard way". It was not enough to only practice ATDD/AAT and explore it from perspective of a software developer; additionally, everything had to be evaluated from other stakeholders' points of view (testers, domain experts, customers) and the outcomes had to be documented in this master's thesis. Although the author is still far from being a true ATDD master, combining empirical experience with working on this master's thesis definitely resulted in thorough and versatile understanding of the subject.

This work and experience of practicing ATDD had also been a rewarding social experience which taught a lot about mechanics of communication between project members. The need to explain and document benefits and drawbacks of ATDD forced the author to look at problems through other people's eyes and to examine the reasons why different stakeholder groups behave in certain ways. Such experience should help in avoiding and overcoming common issues in later projects. For example, ATDD shouldn't be introduced to customers too early or too intrusively. First, a solid test framework should be created. The tests tend to become much cleaner as development goes by, developers become more experienced and similar test patterns emerge. Hence, it's wise to show the tests to customers when the project has been going for a while and when the test framework has become mature enough. The test framework should impress the customers with it's flexibility and ease of use, not to scare them away by looking weird and complex. Unfortunately, currently available ATDD tools are way too hard to use even for programmers, let alone non-technical people who customers usually are. According to this

and several other studies, customers (and even internal domain experts) might be interested in discussing the tests but asking them to edit and create tests themselves is too much.

The most valuable lesson learned throughout this work is that even complex legacy software systems can be made easy to maintain and can be extensively improved without breaking system functionality. Test harness works very well in revealing essential problems even if it is incomplete and covers only the most complex features. Choosing only complex tests scenarios resulted in relatively high amount of small bugs being found during manual exploratory testing but most of them were trivial and very easy to fix. The author became decisively ”test infected” by realizing that programming without a safety network provided by automatic tests is an unproductive, stressful, insecure and disappointing way of working which can even cause depression to worst perfectionists. ATDD is not a silver bullet, but it is definitely an invaluable ally in the battle against software complexity and low quality.

References

[1] Brooks, Frederick P., *No Silver Bullet: Essence and Accidents of Software Engineering*, Computer, Vol. 20, No. 4 (April 1987) pp. 10-19 <URL(4 Dec 2011): `http://sys.cs.rice.edu/course/comp314/10/lectures/Brooks87.pdf`>.

[2] Dr. Winston W. Royce, *Managing The Development Of Large Software Systems*, Proceedings, IEEE WESCON, August 1970, pages 1-9, <URL(4 Dec 2011): `http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf`>.

[3] *The Standish Group CHAOS Report 1995*, The Standish Group, 1995, <URL(4 Dec 2011): `http://www.projectsmart.co.uk/docs/chaos-report.pdf`>.

[4] *Systems Engineering for Intelligent Transportation Systems*, U.S. Department of Transportation, January 2007, <URL(4 Dec 2011): `http://ops.fhwa.dot.gov/publications/seitsguide/seguide.pdf`>.

[5] Larman, C. and Basili, V.R.,*Iterative and Incremental Development: A Brief History*, IEEE Computer Society, June 2003, <URL(4 Dec 2011): `http://dx.doi.org/10.1109/MC.2003.1204375`>.

[6] *CHAOS Summary 2009*, The Standish Group, 2009, <URL(4 Dec 2011): `http://www.portal.state.pa.us/portal/server.pt/document/690692/standish_group_chaos_summary_2009_pdf`>.

[7] *Why Software Fails, Robert N. Charette*, ieee spectrum, September 2005, <URL(4 Dec 2011): `http://spectrum.ieee.org/computing/software/why-software-fails/0`>.

[8] *Wikipedia article about Watts S. Humphrey*, <URL(4 Dec 2011): `http://en.wikipedia.org/wiki/Watts_Humphrey`>.

[9] Watts S. Humphrey, *Why Big Software Projects Fail: The 12 Key Questions*, CrossTalk, March 2005, <URL(4 Dec 2011): `http://www.crosstalkonline.org/storage/issue-archives/2005/200503/200503-Humphrey.pdf`>.

[10] *Agile Manifesto*, <URL(4 Dec 2011): `http://agilemanifesto.org`>.

[11] Watts S. Humphrey, *The Team Software Process (TSP)*, Team Software Process Initiative, November 2000, <URL(4 Dec 2011): `http://www.sei.cmu.edu/reports/00tr023.pdf`>.

[12] Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley Professional, October 5, 1999.

[13] Ken Schwaber, Mike Beedle, *Agile Software Development with Scrum*, Prentice Hall, October 21, 2001.

[14] Ken Schwaber, Jeff Sutherland, *The Scrum Guide*, Scrum.org, October 2011, <URL(4 Dec 2011): `http://www.scrum.org/storage/scrumguides/Scrum_Guide.pdf`>.

[15] Lasse Koskela *TEST DRIVEN, Practical TDD and Acceptance TDD for Java Developers*, Manning Publications Co., 2008.

[16] Takeuchi, Hirotaka; Nonaka, Ikujiro (January-February 1986), *The New New Product Development Game*, Harvard Business Review, <URL(4 Dec 2011): `http://hbr.org/product/new-new-product-development-game/an/86116-PDF-ENG`>.

[17] James Shore, *The Decline and Fall of Agile*, <URL(4 Dec 2011): `http://jamesshore.com/Blog/The-Decline-and-Fall-of-Agile.html`>.

[18] *FitNesse, the fully integrated standalone wiki, and acceptance testing framework*, <URL(4 Dec 2011): `http://fitnesse.org`>.

[19] David L. Parnas, Paul C. Clements, *A rational design process and how and why to fake it*, IEEE Transactions on Software Engineering, Volume 12 Issue 2, February 1986, <URL(4 Dec 2011): `http://users.ece.utexas.edu/~perry/education/SE-Intro/fakeit.pdf`>.

[20] Martin Fowler *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional, July 8, 1999.

[21] Kent Beck *Test-Driven Development By Example*, Addison-Wesley Professional, November 18, 2002.

[22] Esko Luontola, *TDD is not test-first. TDD is specify-first and test-last.*, <URL(4 Dec 2011): http://blog.orfjackal.net/2009/10/tdd-is-not-test-first-tdd-is-specify.html>.

[23] Dan North, *Introducing BDD*, Better Software magazine, March 2006, <URL(4 Dec 2011): http://blog.dannorth.net/introducing-bdd/>.

[24] Tim Mackinnon (Connextra), Steve Freeman (BBST), Philip Craig (Independent), *Endo-Testing: Unit Testing with Mock Objects*, XP eXamined, Addison-Wesley, 2000, <URL(4 Dec 2011): http://www.mockobjects.com/files/endotesting.pdf>.

[25] David S. Janzen, Hossein Saiedian *A Leveled Examination of Test-Driven Development Acceptance*, Software Engineering, 2007. ICSE 2007. 29th International Conference on Software Engineering, <URL(4 Dec 2011): http://dx.doi.org/10.1109/ICSE.2007.8>.

[26] Vesa Lappalainen, Jonne Itkonen, Ville Isomöttönen, Sami Kollanus *ComTest: A Tool to Impart TDD and Unit Testing to Introductory Level Programming*, in proceedings of the fifteenth annual conference on Innovation and technology in computer science education, 2010, <URL(4 Dec 2011): http://dx.doi.org/10.1145/1822090.1822110>.

[27] Roy Osherove *The Art of Unit Testing: With Examples in .Net*, Manning Publications, June 3, 2009.

[28] Uncle Bob (Robert C. Martin), *TDD with Acceptance Tests and Unit Tests*, <URL(4 Dec 2011): http://blog.objectmentor.com/articles/2007/10/17/tdd-with-acceptance-tests-and-unit-tests>.

[29] Michael Feathers *API Design as if Unit Testing Mattered*, Object Mentor, Inc, 2007, <URL(4 Dec 2011): http://objectmentor.com/resources/articles/as_if_unit_testing_mattered.pdf>.

[30] *NUnit, A unit-testing framework for all .Net languages*, <URL(4 Dec 2011): http://www.nunit.org>.

[31] *List of unit testing frameworks*, Wikipedia, <URL(4 Dec 2011): http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks>.

[32] Dan North, *How to sell BDD to the business*, <URL(4 Dec 2011): http://skillsmatter.com/podcast/java-jee/how-to-sell-bdd-to-the-business/>.

[33] *JBehave, Behaviour-driven development in Java*, <URL(4 Dec 2011): http://jbehave.org/>.

[34] Neel Lakshminarayan *BDD is not about tools*, <URL(4 Dec 2011): http://neelnarayan.blogspot.com/2011/04/bdd-is-not-about-tools.html>.

[35] *Cucumber, Behaviour driven development with elegance and joy*, <URL(4 Dec 2011): http://cukes.info/>.

[36] *Concordion, an open source tool for writing automated acceptance tests in Java*, <URL(4 Dec 2011): http://www.concordion.org>.

[37] Robert C. Martin, *Test-driving GUI*, <URL(4 Dec 2011): http://stackoverflow.com/questions/673842/how-to-test-drive-gwt-development/674759\#674759>.

[38] Alex Ruiz, Yvonne Wang Price, *Test-Driven GUI Development with TestNG and Abbot*, IEEE Software, May-June 2007, <URL(4 Dec 2011): http://dx.doi.org/10.1109/MS.2007.92>.

[39] *jfcUnit, an extension to the JUnit.*, <URL(4 Dec 2011): http://jfcunit.sourceforge.net/>.

[40] *Abbot Java GUI Test Framework*, <URL(4 Dec 2011): http://abbot.sourceforge.net/doc/overview.shtml>.

[41] Michael Feathers, *The Humble Dialog Box*, 2002, <URL(4 Dec 2011): http://www.objectmentor.com/resources/articles/TheHumbleDialogBox.pdf>.

[42] *FitNesse, Two-Minute Example*, <URL(4 Dec 2011): http://fitnesse.org/FitNesse.UserGuide.TwoMinuteExample>.

[43] *Digia Financial Systems introduction document*, Digia Plc, 2010, <URL(4 Dec 2011): http://tinyurl.com/6j6ng8e>.

[44] *SLIM, Simple List Invocation Method*, <URL(4 Dec 2011): http://fitnesse.org/FitNesse.UserGuide.SliM>.

[45] *FitNesse ClassPath configuration*, <URL(4 Dec 2011): http://fitnesse.org/FitNesse.UserGuide.ClassPath>.

[46] *FitNesse Decision Table*, <URL(4 Dec 2011): http://fitnesse.org/FitNesse.UserGuide.SliM.DecisionTable>.

[47] Dennis Sosnoski, *Java programming dynamics, Part 2: Introducing reflection*, IBM developerWorks, June 2003, <URL(4 Dec 2011): http://www.ibm.com/developerworks/library/j-dyn0603>.

[48] *SoapUI, a free and open source cross-platform Functional Testing solution*, <URL(4 Dec 2011): http://www.soapui.org/>.

[49] Gayane Azizyan, Miganoush Katrin Magarian, Mira Kajko-Mattson *Survey of Agile Tool Usage and Needs*, AGILE Conference (AGILE), 2011, <URL(4 Dec 2011): http://dx.doi.org/10.1109/AGILE.2011.30>.

[50] Børge Haugset, Geir Kjetil Hanssen, *The home ground of Automated Acceptance Testing: Mature use of FitNesse*, AGILE Conference (AGILE), 2011, <URL(4 Dec 2011): http://dx.doi.org/10.1109/AGILE.2011.37>.

[51] Børge Haugset, Geir Kjetil Hanssen, *Automated Acceptance Testing Using Fit*, HICSS '09. 42nd Hawaii International Conference on System Sciences, 2009, <URL(4 Dec 2011): http://dx.doi.org/10.1109/HICSS.2009.83>.

[52] Børge Haugset, Geir Kjetil Hanssen, *Automated Acceptance Testing: A Literature Review and an Industrial Case Study*, AGILE '08. Conference, 2008, <URL(4 Dec 2011): http://dx.doi.org/10.1109/Agile.2008.82>.

[53] Rick Mugridge, Ward Cunningham, *Fit for Developing Software: Framework for Integrated Tests*, (Robert C. Martin). Prentice Hall PTR Upper Saddle River, NJ, USA, 2005.

[54] James Shore, *The Problems With Acceptance Testing*, <URL(4 Dec 2011): http://jamesshore.com/Blog/The-Problems-With-Acceptance-Testing.html>.

[55] Gandhi, P., et al., *Creating a living specification using Fit documents*, In proceedings of Agile, 2005.

[56] Melnik, G., K. Read, F. Maurer, *Suitability of FIT User Acceptance Tests for Specifying Functional Requirements: Developer Perspective*, In proceedings of XP/Agile Universe, 2004.

[57] Ricca, F., M. Di Penta and M. Torchiano, *Guidelines on the use of Fit tables in software maintenance tasks: Lessons learned from 8 experiments*, Software Maintenance, 2008. ICSM 2008. IEEE International Conference on 2008.

[58] Melnik, G., *Empirical Analyses of Executable Acceptance Test Driven Development*, in PhD Thesis, Department of Computer Science, 2007, University of Calgary: Calgary, Alberta. p 190.

[59] Melnik, G. and F. Maurer, *Multiple Perspectives on Executable Acceptance Test-Driven Development*, In proceedings of XP, 2007.

[60] Deng, C., P. Wilson and F. Maurer, *FitClipse: A Fit-based Eclipse Plug-In for Executable Acceptance Test Driven Development*, in Proceedings of XP, 2007.

[61] SeleniumHQ, Web application testing system, <URL(4 Dec 2011): `http://seleniumhq.org`>.

[62] Robot Framework, A generic test automation framework, <URL(4 Dec 2011): `http://code.google.com/p/robotframework`>.

[63] Holmes, A. and Kellogg, M., *Automating functional tests using Selenium*, in proceedings of Agile Conference, 2006.

[64] Ricca, F., et al., *Using acceptance tests as a support for clarifying requirements: A series of experiments*, Information and Software Technology, 2009. 51(2): p. 270-283.

[65] Ricca, F., et al., *Are Fit Tables Really Talking? A Series of Experiments to Understand whether Fit Tables are Useful during Evolution Tasks*, in Icse'08 Proceedings of the Thirtieth International Conference on Software Engineering. 2008. p. 361-370.

[66] Geras, A., et al., *A Survey of Test Notations and Tools for Customer Testing*, in proceedings of Extreme Programming and Agile Processes in Software Engineering. 2005.

[67] Melnik, G., F. Maurer, M. Chiasson, *Executable acceptance tests for communicating business requirements: customer perspective*, In proceedings of Agile Conference, 2006.

[68] Park S. and F. Maurer, *Communicating Domain Knowledge in Executable Acceptance Test Driven Development*, in Agile Processes in Software Engineering and Extreme Programming. 2009. p. 23-32, <URL(4 Dec 2011): `http://dx.doi.org/10.1007/978-3-642-01853-4_5`>.