Masterarbeit

Aspects of Test-Driven Development

Pulkit Chouhan

November 2013

Betreuer: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany
Begutachter: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Technische Universität Graz
Institut für Softwaretechnologie

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am ……. ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

……. ……..
date (signature)

Acknowledgements

I would like to thank Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany for his support and patience during all the time I was writing this thesis. Furthermore thanks to the company BCT Electronics in Salzburg, Austria who allowed me to work on the practical part of this thesis during my normal working hours.

Also I would like to thank my family for all their support during the whole period of my studies because without them none of this would have been possible.

Finally, thanks to my friends wo always kept me motivated.

Kurzfassung

Testgetriebene Softwareentwicklung hat sich in den letzten Jahren zu einer weit verbreiteten und angesehenen Form der Entwicklung für Systeme aller Art entwickelt. Die Anwendung der testgetriebenen Entwicklung macht jedoch isoliert von anderen Praktiken der agilen Methoden wenig Sinn und auch das volle Potenzial lässt sich auf diese Weise bei weitem nicht ausschöpfen. Diese Arbeit versucht einen Überblick darüber zu geben, warum traditionelle Herangehensweisen oft scheitern und wie man die testgetriebene Entwicklung in ein Umfeld einbetten kann, um das Design zu verbessern und die Fehlerhäufigkeit zu senken. Dazu werden Prinzipien der agilen Methoden vorgestellt und auf die Werte eingegangen die dabei wichtig sind. Es wird erläutert wie Tests zu organisieren und zu schreiben sind, damit sie ihre Dokumentations- und Spezifikationsfunktion erfüllen können. Weiters wurde eine Applikation unter den Gesichtspunkten der testgetriebenen Entwicklung hergestellt und wird in dieser Arbeit kurz näher vorgestellt.

Abstract

Test-Driven Development has been around for several years and has become a widely known and accepted form of software development for all kinds of systems. Applying Test-Driven Development isolated from other practices of agile development will not harvest from the full potential of this development methodology. This thesis tries to give an overview of the reasons why traditional development processes often fail and how to integrate the test-driven approach into the net of core agile practices to create a simple design and tests that act as documentation and specification. Principles of test organisation and patterns of test structure are explained. Furthermore, a application was developed following the principles of Test-Driven Development and will be briefly described in this paper.

Contents

List of Figures

List of Tables

Abbreviations

XP	eXtreme Programming
TFD	Test-First Development
TDD	Test-Driven Development
ATDD	Acceptance Test-Driven Development
BDD	Behavior-Driven Development
SUT	System Under Test
CBO	Coupling Between Objects

1 Traditional Process Models in Software Development

Traditional process models for developing software (or systems) have been around since the 1950's. Scacchi distinguishes two kinds of heavyweight life cycle models(Sca01, p. 3):

Descriptive Models are used to describe the past. They document how a system was developed and are used to gain understanding and to extrapolate improvements for software development models.

Prescriptive Models defines how a new (software) system should be developed. They establish 'guidelines or frameworks to organize and structure how software development activities should be performed, and in what order'(Sca01, p. 3).

Both kind of models help organizing the development process of software. This includes the planning and the scheduling of work in regards of staff, budget and other resources, choosing the right methodologies to assure software quality.(Sca01, p. 3)

The term heavyweight is an indicator for the 'degree of formalization of the processes and the number of associated (intermediate) results or (intermediate) prod-

ucts'. (Fin06, p. 90)

Furthermore, Scacchi identifies a number of activities (or a subset) that most of the existing life cycle models have in common(Sca01, p. 1-3). The most common are:

System Initiation/Planning New systems should replace, extend or improve existing solutions.

Requirement Analysis and Specification This phase identifies the problems that should be solved by the implemented system (functional requirements) but also its performance, resource and maintenance characteristics.

Partition and Selection Partition given requirements and specifications into work-packages which possibly define logical subsystems. For each subsystem, decide whether to buy, reuse or implement it.

Architectural Design Defines the overall design of the whole system and also the interfaces interconnecting the subsystems.

Detailed Component Design Specification The design of the submodules is defined during this phase.

Component Implementation and Debugging The previously defined specifications are transformed into source code and their basic functionality is validated.

Software Integration and Testing In this phase, the overall functionality and integrity of the implemented system and subsystems is tested against the previously defined requirements (functional and non-functional) and architecture.

Documentation Revision and System Delivery Create documents that describe the system development and allow system support.

Deployment and Installation The implemented system is deployed, installed and configured (access privileges, diagnostic tests).

Training and Use System users are provided with training and guidance for an effective usage of the installed system.

Software Maintenance Provide patches to fix bugs, enhance functionality and improve performance.

As (Fin06, p. 89) points out, each phase may only be entered, if the predecessor phase has been finished and approved. This makes it necessary to well-define the results needed for the completion of a phase.

The following subsections will focus on some heavyweight, prescriptive life cycle models to finally discuss the issues that led to the development of agile methodologies.

1.1 The Code-and-Fix Model

This model, already used in the earliest days of software development, basically consists of only two steps (Boe88, p. 61):

- The actual coding
- Find bugs and fix them

Even the designing phase comes after writing the code. As (Boe88, p. 62) denotes, this eventually leads to poorly structured code, especially after fixing some bugs. This flaw makes bugs found late during development very expensive to fix. And even if the software was well designed, it often failed to match the customer's needs. So it was rejected, showing the need for a requirement analysis phase prior to the design phase(Boe88, p. 62).

1.2 The Waterfall Model

The Waterfall Model, also known as the classic software life cycle or the Linear Sequential Model, was first described by Walter Royce in 1970 (Gus02; PWB09; Sca01). Compared to the Code-and-Fix Model, it introduced two important improvements(Boe88, p. 63):

- Feedback loops and guidelines on how to apply them to minimize rework
- Some kind of prototyping

(Sca01, p.5) denotes that the waterfall model resembles a finite state machine. This is due to the fact that the evolution of the software happens through the transition from one phase to another. A lot of projects realized by using the waterfall model go through the phases Requirements Engineering, Design and Implementation, Testing, Release and Maintenance(PWB09, p. 4).

Only if a phase (i.e. the documents belonging to this phase (Boe88, p.63)) passes previously defined requirements, the evolution continues and the development moves to the next phase. In practice although, often one or more phases tend to overlap and there even is feedback from one phase to another, turning the waterfall model into a non-linear process model. (Som11, p. 31)

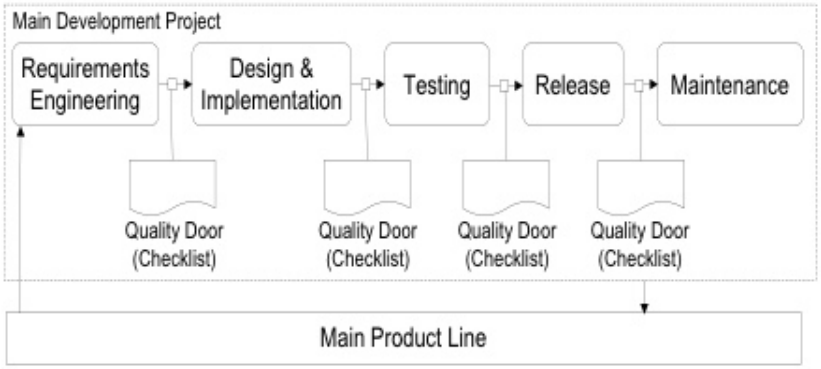

Figure 1.1: The Waterfall Model (PWB09, p. 4)

1.3 Issues when applying the Waterfall Model

In his paper, (PWB09, p. 2) argues that according to some studies about the waterfall model, one of the main reasons for failing projects is the management of a large scope. As pointed out in (Som11, p.32), another difficulty is the integration of the final product and the enormous amount of testing. Because of it's inflexibility, commitments must be made at a very early stage during the development process. This life cycle should only be chosen if the requirements are well defined, understood and unlikely to change. (Som11, p. 32)

Testing is done late when applying the waterfall model. Late testing leads to a high amount of hard to fix bugs and to reduced code coverage. This late feedback then leads to quality issues. (PWB09, p. 11)

(Boe88) too emphasises on the difficulties caused by the the fact that all documents belonging to a phase have to fulfill all completion criteria before finishing a phase. This causes problems especially in some early and important phases like requirement analysis and design, because at this stage, the requirements are often poorly understood but never the less the phase has to be completed before going

on to the next phase. Especially for end-user applications this drawback could be fatal, as requirements keep changing a lot. (Boe88, p. 63)

Table 1.1 shows an overview of some of the issues identified by (PWB09).

ID	Issue
1	High effort and costs for writing and approving documents for each development phase.
2	Extremely hard to respond to changes.
3	When the system is put to use the customer discovers problems of early phases very late and system does not reflect current requirements.
4	Problems of finished phases are left for later phases to solve.
5	Management of a large scope of requirements that have to be baselined to continue with development.
6	Big-bang integration and test of the whole system in the end of the project can lead to unexpected quality problems, high costs, and schedule overrun.
7	Lack of opportunity for customer to provide feedback on the system.
8	The waterfall model increases lead-time due to that large chunks of software artifacts have to be approved at each gate.
9	When iterating a phase the iteration takes considerable effort for rework.

Table 1.1: Issues in Waterfall model (PWB09, p. 3)

1.4 The Spiral Model

The spiral model (see figure 1.2) is a so called risk-driven software process framework first proposed by Boehm in 1988, trying to incorporate the strengths of some other models and avoiding their drawbacks. (Boe88, p. 61)

The idea behind this model is that each cycle of the spiral represents a process

(i.e. sequence of steps) that has to be applied to each level of elaboration of the product (Boe88, p. 61). The radial dimension "represents the cumulative cost incurred in accomplishing the steps up to date"(Boe88, p. 65) and the angular dimension "represents the progress in completing each cycle of the spiral."(Boe88, p. 65)

Each quarter of the model stands for a ordered phase. At the beginning of each cycle, it is necessary to define the objectives to work on and how to implement those objectives (first quarter). Each alternative way of realising an objective needs to be evaluated in regards of uncertainty leading to the identification of sources of project risks. Evaluating and finding solutions for minimizing those risks is the next step of the cycle (second quarter).(Boe88, p. 65).

(Som11, p. 49) describes the next two steps as follows:

Development and Validation The third phase of a development cycle (or the third quarter) consists of choosing and applying a development model (e.g. throwaway prototyping, development based on formal transformations, ...).

Planning In the final phase, the current state of the project is reviewed if further development is necessary, the next cycle is planned.

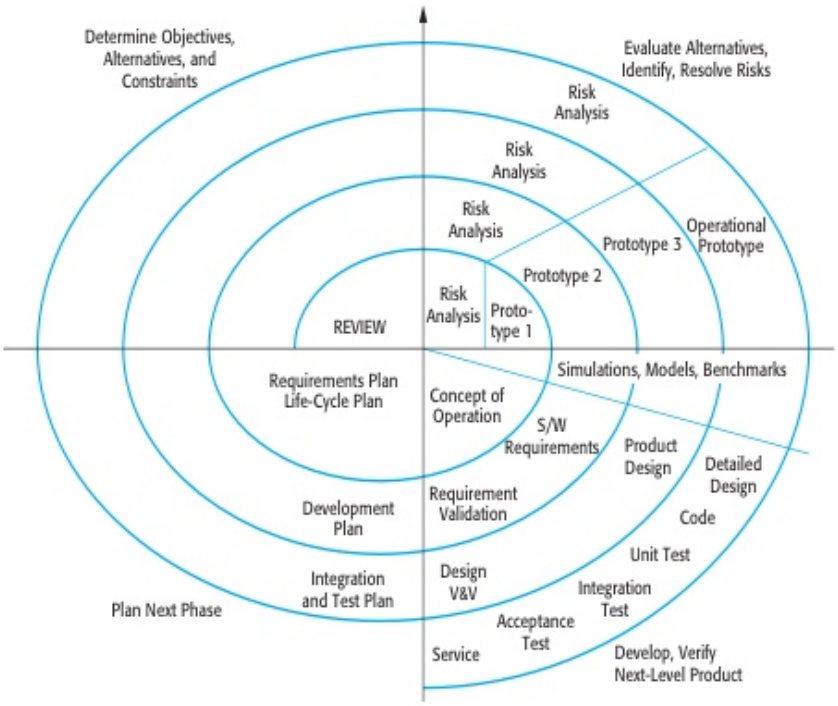

Figure 1.2: The Spiral Model (Som11, p. 49)

2 Test-Driven Development

As the previous chapter showed, it has become neccassary to establish a new way of software development to ensure high quality of the product while keeping the developers motivated.

2.1 Agile Software Development

As already showed, the inflexibility of the traditional development models made it necessary to come up with a solution that provided the missing flexibility but still remained structured and methodological.

For this reason, in February 2001 seventeen experienced software developers[1] gathered in Utah to define common principles of software development from a agile perspective which they called the agile manifesto (see figure 2.1).

Individuals and Interactions over Processes and Tools According to (HD08, p. 5), this principle's message is to focus on the people involved in the development of an system rather than investing a lot of effort into adapting to new methodologies and tools. The way a team (i.e the members of that team)

[1] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas


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.

Figure 2.1: The Agile Manifesto (HD08)

communicate, interact and discuss will have a high impact on the project's success and is therefore of very high priority.

Working Software over Comprehensive Documentation Only those documents should be created, which are essential to the development process. The created documents have to be accessible for all stakeholders involved in the collaborative workspace. Furthermore, the coding should begin as soon as possible to gain domain knowledge (not only for the developers, but also for the customer). As one can see, the coding at a very early stage is contradictional to heavy weight development models with high amount of process ceremony as it is essential for them to document and understand all the requirements before coding, to ensure quality (HD08).

Customer Collaboration over Contract Negotiation Close contact to the customer is embraced to allow them to understand and support changes that come along with software projects. Also, this interaction will encourage developers to base their work on the information provided by the customer (HD08).

Responding to Change over Following a Plan Agile development provides an context that allows all stakeholders of a project to cope with changes that occur during development, enabling the team to still deliver high quality. This pillar of the agile manifesto is based on the fact that even customers don't know all of the requirements at the beginning of the development of a product. Requirements are gradually refined and better understood as the development evolution continues (HD08).

The agile manifesto, which is backed up by twelve underlying principles, can be applied to different agile methods such as: Scrum, Extreme Programming, Feature-Driven Development etc. (HD08). The principles are as follows (BFC09):

- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

- Continuous attention to technical excellence and good design enhances agility.
- Simplicity–the art of maximizing the amount of work not done–is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

2.1.1 The Agile Team

The agile team is a paramount part of the context in which the process of Test-Driven Development should be embedded. This section will focus on this aspect, giving an overview of the roles played by the members, their rights and responsibilities. It will also show how teams should be composed to support agile development.

2.1.1.1 Roles in an Agile Team

One key factor of each agile development method are the roles specified by the method. Some others are practices, values, techniques and tools(DH04, p. 157). (AL12, p. 61) defines a role as a part a team member plays in a given situation (e.g. team lead, stakeholder etc.). Along with the role come rights that can be claimed by the member assigned that role. Finally, a role assigns specific responsibilities to each team member. The team member holding a role does not have to execute all the activities related to his or her role. Those activities can be delegated to somebody else. The responsibility of the role holder is to ensure that the delegated aspect of the role is carried out properly (HD08, p. 29). Furthermore, there are

some rights and responsibilities that each person involved with a agile projects has regardless of the roles he or she has been assigned. A subset of those are (AL12, p. 63):

- The right to
  - be supported
  - decide how to invest one's resources
  - have the estimates for the required activities respected
  - be treated with respect
  - have a safe working environment
- The responsibility to
  - optimize the usage of resources
  - guide and coach other team members
  - avoid and deny work outside the current iteration
  - share information

Roles are not the position of a team member inside the organization. A person with the role of a stakeholder could be the head of department outside the project (AL12, p. 63). Roles often found in agile teams are the stakeholder, product owner, team member, team lead, architecture owner (AL12, p. 66). These roles are also called primary roles (see figure 2.2).

Stakeholder A stakeholder is everybody affected by the outcome of the project:

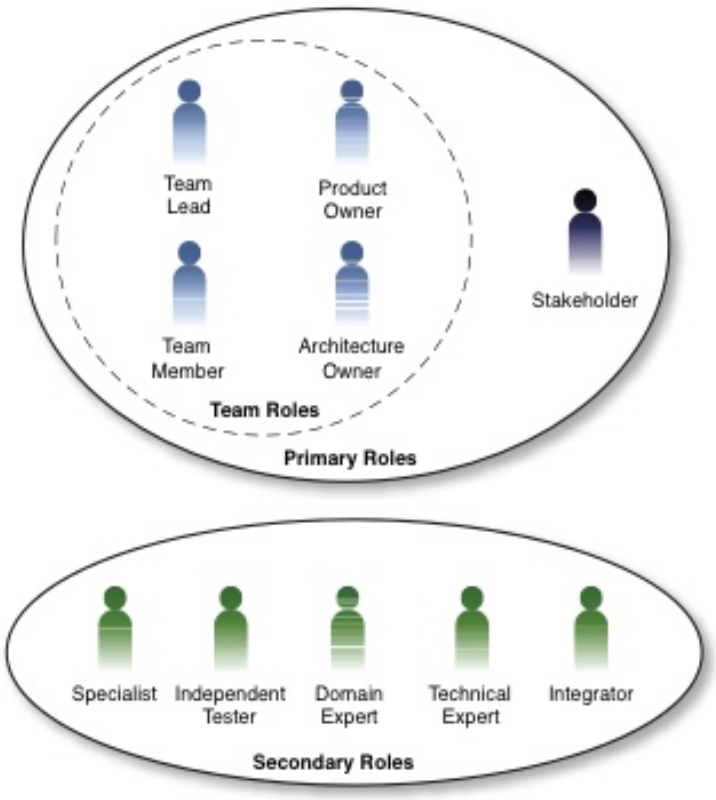

Figure 2.2: The Roles in an Agile Team (AL12, p. 66)

users, senior manager, support, etc.. Generally stakeholders can be divided into groups of end users, principals (they pay for the system and put it to use), partners (operational staff) and insiders (people from the development team). (AL12, p. 67)

Product Owner The product owner is responsible for representing the needs and desires of the stakeholders (the customer) to the development team and manage a list of work items which will be realised to implement the product. The product owner reacts as the interface between the agile team and the stakeholders (AL12, p. 67). Also, this person hast to present the finished work to the stakeholders.

Team Member Everybody with this role assigned will work on testing, analysing, designing, programming and planning activities throughout the project. (AL12, p. 68)

Team Lead The main responsibility of the team lead is to enable the team members to perform technical management activities, acting as a servant-leader and allowing them to self organize as a team. He or she acts as a agile coach helping the members to deliver results for the commitments made with the product owner. (AL12, p. 73)

Role	Description
Leading Group	
Coach	Coordinates and solves group problems, checks the web forum and responds on a daily basis, leads some development sessions.
Tracker	Manages the group diary, measures the group progress with respect to the estimations and tests score, manages and updates the boards.
Customer Group	
End user	Performs on-going testing of the software as an end user, contacts real end users to test the software, collects and processes the feedback received.
On site customer	Tells customer stories, makes decisions pertaining to each release and iteration, provides feedback, defines and develops acceptance tests.
In charge of acceptance testing	Works with the customer to define and develop acceptance tests, learns the topic of first-test development and instructs the others on the subject.
Code Group	
In charge of unit testing	Learns about unit testing, establishes an automated test suite, guides and supports others in developing unit tests.
In charge of design	Maintains current design, works to simplify design, searches for locations in the software that need refactoring and ensures proper execution of such.
In charge of code standards and tools	Establishes and refines group code standards, searches for development tools that can help the team, guides and supports in the maintaining of standards and use of tools.
In charge of code effectiveness and correctness	Guides other teammates in the benefits of pair programming, enforces code inspection in pairs, searches for places in the code whose effectiveness requires improvement.
In charge of continuous integration	Establishes an integration environment including source control mechanism, publishes rules pertaining to the addition of new code using the test suite, guides and supports other teammates in the integration task.
Maintenance Group	
In charge of presentations	Plans, organizes and presents version presentations, demos, and time schedule allocations.
In charge of documentation	Plans, organizes and presents the project documentation: process documentation, user's guide, and installation instructions.
In charge of installation shield	Plans and develops an automated installation kit, supports and instructs other teammates as to the appropriate way to develop software for easy and correct installation.

Figure 2.3: The Roles in an Agile Team according to (HD08, p. 162)

(HD08) identified finer granulated roles with very similar responsibilities (see figure 2.3).

2.2 Introduction to Test-Driven Development

Test-Driven Development is a methodology based on the principle of Test-First Programming. It first appeared as a part of Kent Beck's eXtreme Programming approach and is still used heavily in the XP context. Test-First Programming (see Figure 2.5) means writing functional tests on a very low level before writing actual production code, giving developers instant feedback on how new functionality (or any changes for that matter) interferes with existing code (EMT05). Furthermore, the Test-First philosophy provides some other advantages as (EMT05):

Task-orientation Forcing developers to spilt a problem into smaller, manageable parts, providing gradual and measurable progress.

Quality Assurance Often running the developed tests gives the development team the security of having a executable product at any time

Low-level Design Unit tests drive the decision which classes and methods need to be created, named and the interfaces they offer.

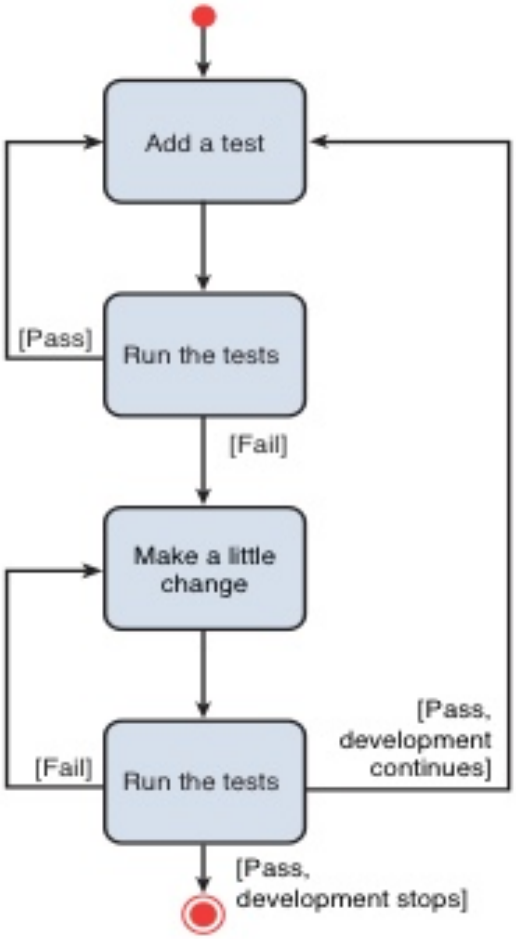

Figure 2.4: The Test-First Development Model (AL12, p. 349)

(Mes07) denotes that if tests are written after production code, the likelihood of not having to change the production code is very low because the defects are found very late in the development cycle. It is also very challenging to write tests after developers think they have finished their work. With the test-first philosophy however, the product is designed for testability inherently because the the code developed is written to past the test. Some other advantages are that only enough code is written to pass the test resulting in minimalistic number

of line of code. Also, verification is more naturally when following the test-first paradigm. For example instead of using string representations for comparing the state of a object, one would evaluate the correct attributes of that object instead.

Beck (Bec02, p. 127) denotes that no programmer will test after writing production code. This has to do with what he calls ‘no time for testing‘ death spiral. Pressure coming from above is connected negatively to the process of writing tests. Writing tests on the other side too is connected negatively to stress from above. This means that a developer who does not write tests from the beginning will get more pressure because his code is not working as expected. But if code does not work as expected, pressure will rise, leading to even less tests because the developer will have to fix the old problems first.

If the developer adheres to the test-first paradigm however, writing tests before writing the production code will assure that that the code delivered will pass the tests, reducing pressure from above.

Test-Driven Development extends the idea of Test-First Programming by adding the activity of refactoring (AL12, p. 349). Refactoring is seen as a structured way to improve code quality over time by applying small changes, retaining the semantics of the business logic(AL12, p. 349).

2.2.1 Fearless Development

As (Bec00) mentions, Test-Driven Development is a way not to terminate fear completely but to reduce and manage it. He calls Test-Driven Development a technique to be aware ‘of the gap between decision and feedback during programming‘ (Bec00). Instead of being tentative, grumpy, less communicative and shying away from feedback, Test-Driven Development helps developers to learn quickly,

communicate more clearly and look out for feedback. In their article (JM07) denote the fact, that no developer creates code without bugs. Especially when first writing the production code and later testing manually. When working with legacy systems that grew over time, developers often tend to avoid even minor changes due to the fact that those systems are often poorly understood and even small changes could lead to a unpredictable house of cards. (JM07)

Test-Driven Development fights this uncertainty by a growing collection of tests, called the regression suite. This allows developers to become more confident in the code and so be more relaxed as each line of code is (ideally) backed up by tests. When tests are written after coding, developers tend to give only a shallow look on what actually broke the code, making testing a dull and boring activity. (JM07). When working with Test-Driven Development on a higher lever (called Acceptance Test-Driven Development) this methodology helps finding requirements, improve communication and bring clearance (JM07).

Design is improved due to the fact that writing tests first forces the developer to think about the new capability he or she is going to implement in regards of the interfaces, input, usage and behaviour (JM07).

A further fear reducing aspect of Test-Driven Development is the fact that the software grows not revolutionary but evolutionary, taking the burden from the developers shoulder to completely understand the system that is going to be implemented. Instead, at the beginning, it is sufficient to have and understand a simple architecture and a simple design to finish the early iterations and have a functioning prototype as mentioned in section 2.1 (JM07). (Bec00) denoted, that the smaller the evolution is (i.e. the less code a unit test covers) the better a developer understands what he or she is implementing.

2.2.2 Behavior-Driven Development

(AL12) identifies two levels of Test-Driven Development. Behavior-Driven Development and as the second level traditional developer Test-Driven Development. Behavior-Driven Development focuses on JIT specifying of detailed requirements and their validation. Developer Test-Driven Development aims to provided a detailed design and implementation of those specified requirements of the first stage and validate the implementation. Behavior-Driven Development is also known as (Adz11):

- Agile Acceptance Testing
- Example-Driven Development
- Behavior-Driven Development
- Specification By Example

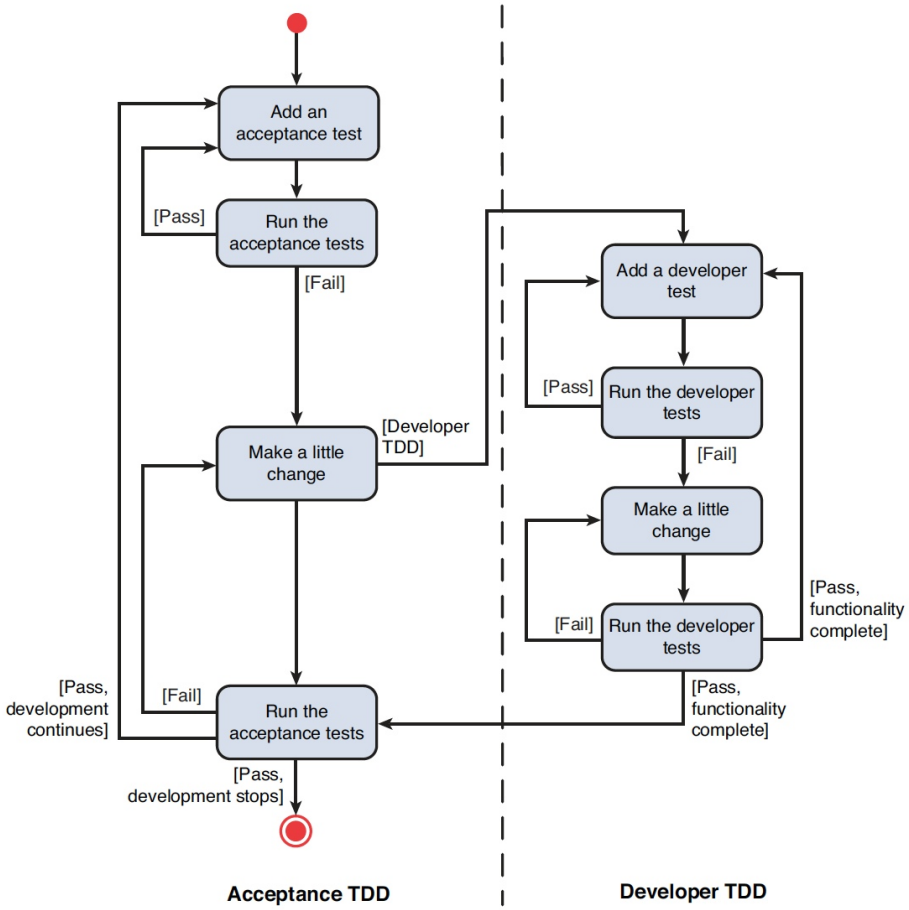

Figure 2.5: Two levels of Test-Driven Development (AL12, p. 351)

BDD helps answering the question 'How do i know when I am done' with a specific user story or a feature(AL12, p. 351). (Rog04) defines acceptance tests as tests that are owned and defined by the customer to verify the completeness and correctness of a user story. Behavior-Driven Development is seen as an effective communication tool due to it's nature because the tests are written by the customer. As development continues, developers start using the acceptance tests as the team's common domain language. Because the acceptance tests (also known as customer tests) need to be understood before implemented, they assure on

going communication between the customer and the development team thereby improving clearance and refinements of the requirements (Rog04).

Through automation, BDD can become part of regression testing, just like the tests of developer Test-Driven Development.

2.3 The Typical Test-Driven Development Cycle

Beck (Bec02) defines the ‘rhythm‘ of Test-Driven Development as a list of 6 activities that need to performed again and again. These activities are:

- Quickly add a test
- Execute all tests and see how the new one fails
- Adapt the production code
- Execute all tests again and if they pass go to the next step. Else, go to the previous step
- Refactor to remove duplication and run tests to see if anything is broken

The test added in the first step should reflect what the developer expects from the new pieces of code. The test is supposed to tell a story and the interfaces used in the test only describe how the developer wishes them to look like. It is important to remember, that the tests are allowed to change and be refactored as the whole system grows. Change, in fact is welcomed and embraced. The interfaces used in tests will be adapted when a simpler or more robust version of them makes sense. When the tests are executed they will fail. Simply because the interfaces and classes referenced in the test might not even exist yet. The goal of the third

activity is to make the tests pass as soon as possible. For that, the developer might use some hard coded test stubs that return the desired value. After all the tests pass, it is time to start writing the real code, remove duplications, refactor. During the whole process, the developer is backed up by the tests. Test-Driven Development is in some cases the opposite of the more formal, architecture-driven approach, where designing and planning first, before writing any code, has the highest priority. (Bec02, p. 11f)

2.3.1 Refactoring

Refactoring is one of the main practices in Test-Driven Development. It helps changing and improving the design of a system towards simplicity and clearness by removing duplication and complexity. The aim is to perform small steps which in the end might sum up to a whole new design. Structures (loops, branches, methods and even classes) that are similar, can be made identical to eliminate one of them. One should always start the refactoring by isolating the part that has to change. One possible way to do that is by extracting code into an extra method, an object or a method object. Of course, the developer will have to write a test first before performing these extractions. Refactoring by isolation and extracting will lead to a higher abstraction level, decreasing the complexity. (Bec02, p. 181f)

When noticing control flows that are getting too complex (for example by too many method calls), the developer might consider not calling the method, but instead perform the steps of the method at the position where it is called. A method that has a very complicated signature could be instead turned into a object (method object). For this, a class is created with the parameters of the method as fields. After creating the method object, the execution of the object's

encapsulated logic is triggered at any point. (Bec02, p. 185ff)

2.3.2 Emerging Design

The aim of software development is to create a system composed by coherent, flexible components that work also in larger environments. By achieving that, it is possible to make the system adaptable to new requirements. Test-Driven Development supports the developer to create such a system by forcing him to write the test first and not think too much about how exactly the implementation will look like. If the intention of the test is unclear and hard to read, the requirements and the bigger picture of the system have not been understood yet properly. By keeping the tests simple and understandable (see 2.5), the scope of each test is automatically limited. Another factor that helps to decouple the components is that the dependencies for a test must be passed on to it. This means, if a very complex fixture (see 2.5.1) is needed to run a test, then the context dependency of a component might be too high, making it less coherent. (FP10, p. 57)

(FP10, p. 58) suggest that with Test-Driven Development, the interface that the components use to communicate with each other are far more important than the class structure. The interfaces allow the objects to collaborate and deliver the required functionality. The main goal during Test-Driven Development should be a design, that heavily enables maintaining. Subsequently, as the system grows, code must be distributed over objects, packages, programs and even systems. All distribution and splitting must be backed up by tests, that verify the correct functionality of the system. There are two heuristics that are used in Test-Driven Development to accomplish the structuring of code. (FP10, p. 47f)

Separation of Concerns As little code as possible should be changed when one tries to alter the systems behavior. This is the main reason to keep code

that performs similar tasks and has similar responsibilities together as close as possible. This pattern will allow the developer to find the important places for code changes quickly. (FP10, p. 48)

Higher Levels of Abstraction To avoid duplication and decrease complexity, a higher level of abstraction is needed. Functionality and responsibilities should be combined instead of implemented twice and distributed over a large number of components. (FP10, p. 48)

If these two concepts are applied strictly, they will cause the system to have an 'anticorruption layer'. This means that the parts responsible for the business domain are separated and isolated from the outside world (the dependencies). The business domain components are connected by so called bridges. These are the interfaces (and the interfaces' implementation) which the components use to communicate with each other. (FP10, p. 48)

2.4 Impact of Test-Driven Development

It is widely assumed that Test-Driven Development leads to less faulty, maintainable, less coupled and better documented code (JS08, p. 77). This section will focus on how Test-Driven Development influences design, architecture and the quality of software products and will look into the evaluation of this software development practice. There are some misconception about Test-Driven Development. The first one is that Test-Driven Development is equal to automated testing. And the second one, that Test-Driven Development means writing all tests upfront. Which is not true. When applying the test-first approach, one only writes small, rapid tests for the components being developed at the moment (JS08, p. 77). Tests like system and integration tests still need to be written

afterwards. The main goal of Test-Driven Development is to improve the design and architecture of software, assuring each developer that the changes made on the source code level did not brake the current design. It is a development strategy, not a testing strategy. By having to write the tests first, a developer is forced to think about the interfaces and the business logic before writing code (JS06, p. 1).

2.4.1 Test-Driven Development and the Influence on Quality of Software Design

Ideally, software should be easy to modify, maintain, enhance and reuse. Those are the internal quality indicators of software design (JS06, p.2). An external factor is for example the defect rate. (JS06) set up a number of hypothesis about the Test-First and the Test-Last approach and conducted a experiment to test them. They created three teams of developers. One following the the Test-First approach and the other one following the Test-Last approach. The last team did not write test at all. As a result the Test-First Team (TFT) was able to finish far more requested features than the Test-Last team (TLT) and the No-Test Team (NTT) and even complete a graphical user interface. At the same time TFT did not invest most effort into developing the features. Generally speaking, they had to spend less time on a line of code then the other two teams. (JS06, p.4)

When looking at the code size and test density, the results in (JS06) display significant differences in the amount of line of codes per method and also in the amount of code per feature. The NTT had to write far more code to provide the same functionality and features as the TFT and TLT. (JS08) come to very similar results regarding the number of line of code per module, method and per class (see figure 2.6).

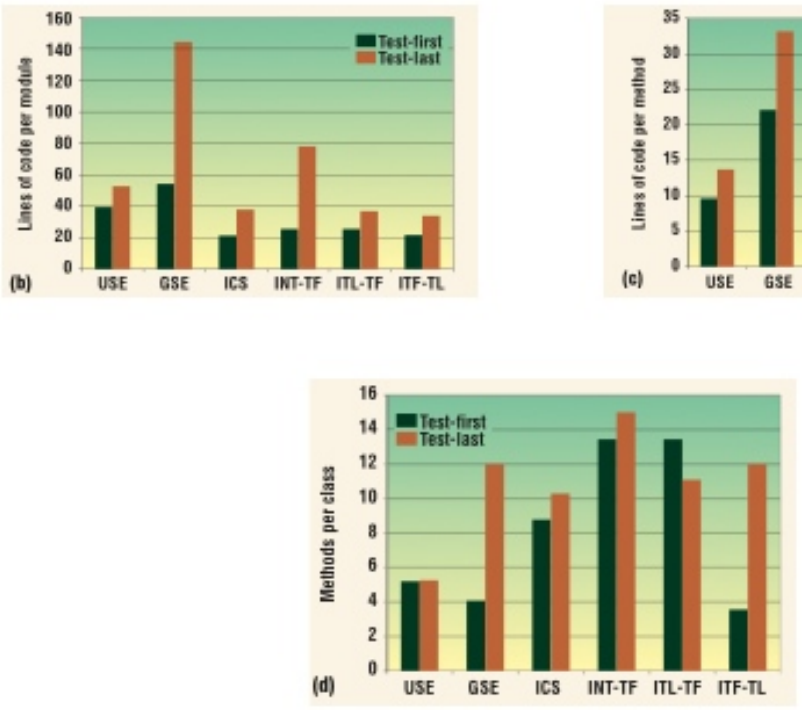

Figure 2.6: Lines of Code per Module, Method and Class (JS08, p. 80)

They denote, that in each study, the test-first approach resulted in in smaller modules, methods and classes, decreasing the complexity of code and thereby making it easier to read and maintain (JS08, p.82). When comparing test coverage, the Test-First approach will lead to a higher rate of code coverage by unit tests and to more assertions than the Test-Last approach (JS06, p.4).

Systems developed with no tests at all or with tests written last, tend to be more procedural than systems developed with the Test-First approach. Test-Last code tends to have a lot of business logic stuffed into a single class and into single methods which results in increasing complexity. Test-First systems on the other side, have the tendency to distribute responsibility among several classes and methods, leading to object-oriented design. (JS06, p.4)

Complexity can be measured in independent paths through code (cyclomatic complexity), nested block depth and CBO. Another unit for complexity is the amount of parameters for methods. Test-Driven Development often results in systems with a higher amount of classes than systems developed without the Test-First paradigm. There are concerns that this fact could lead to higher CBO rates. But no significant CBO higher values could be found in several studies. (JS06, p.4),(JS08, p.82)

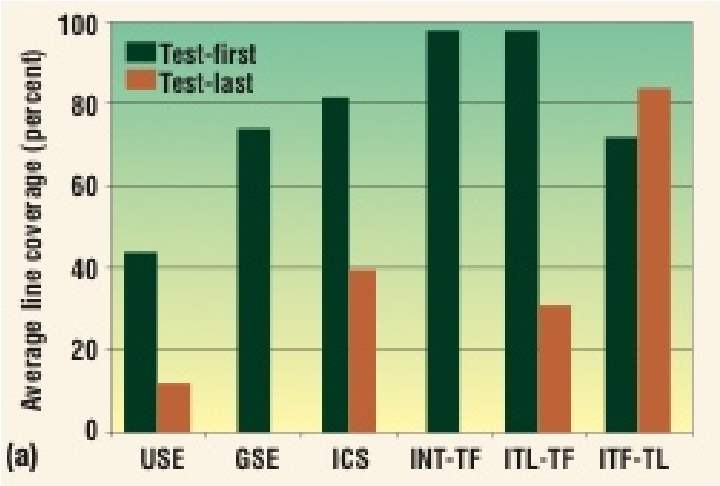

Figure 2.7: Average line of code coverage (JS08, p. 80)

The same holds for the nested block depth and the other metrics, which leads to the assumptions that test-first programmers write simpler and better structured code. The programmers themselves seem to have similar perceptions about Tet-Driven Development. A vast majority of them report that they think the Test-First approach leads to simpler designs and fewer defects. They have higher confidence in the software they developed (JS06, p.7). This fact helps reducing fear during development, which is a major part of the Test-Driven Development paradigm (see 2.2.1).

2.5 Unit Tests and Their Organization, Structure and Patterns

As described in the previous sections, the heart of Test-Driven Development are the unit tests. This section will focus on the attributes of unit tests and also on how they can be organized in regards of structure and patterns. Tests are created and maintained to provide a better understanding of the SUT, improve overall quality and to reduce risk. They should be easy and fast to run, write and maintain (Mes07).

Figure 2.8 shows the typical effort and the return of investment of automated tests during the development of a software system. One can see, that the initial effort of introducing Test-Driven Development is high as the developers might have to learn new techniques but at the end this methodology provides large effort saving potential, given that the tests are maintained and organized in a structured way.

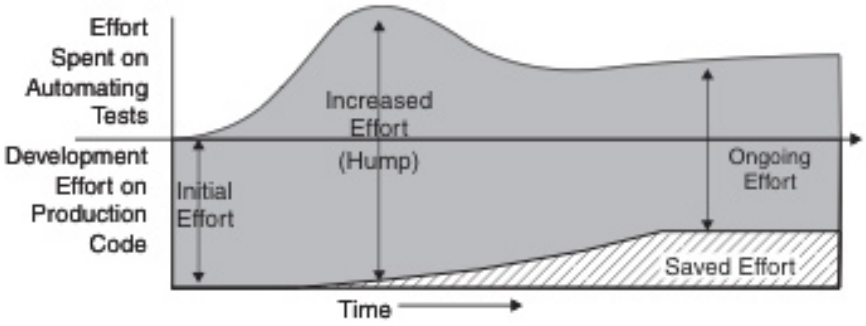

Figure 2.8: Return On Investment after introducing Test-Driven Development (Mes07, p. 20)

In the context of Test-Driven Development, tests represent also the documentation of the system being developed. As (Mes07) mentions, tests provide the

developer (or any other stakeholder for that matter) with a understanding of the system for which otherwise they would have to spend a lot of time reviewing and introspecting the code. The simple reason for this fact is that unit tests describe what a system or a part of the system does by specifying what the result should be but not necessarily *how* the system does it.

Without proper automated tests (or at least some kind of tests) it is impossible to say if a change that has been introduced did break something. This is often an issue with legacy systems. The more code coverage tests provide the smaller is the possibility of breaking something without noticing it. And if the modification broke something, tests provide information where the defect is located. But one has to be careful not to cause harm by creating tests. This could happen when for example test logic is embedded in production code (see 2.5.2). If done correctly, unit tests and therefore Test-Driven Development will lead to, or at least support, fearless development (see 2.2.1). (Mes07)

Unit tests should adhere to some basic rules (Mes07). These are:

High Degree of Automation The higher the degree of automation is, the easier the tests can be run.

Self checking Developers are informed when a test fails, reducing the manual effort if everything passes.

Repeatable Each time a test is run under the same circumstances, the result should be identical.

Expressive The intent of the person who wrote the test should be clear.

Separation of concerns As mentioned previously, test logic and production logic must be kept separately at all times. Furthermore, each test should only

focus on a single concern, allowing a better location of defects.

2.5.1 The Four Phases of an Unit Test

(Mes07) identifies four typical phases that that each unit test should have. These are as follows

- Fixture Setup
- Exercise SUT
- Verify result
- Fixture teardown

Fixture setup is the phase where is the context for the test is created and configured. This environment is called fixture. This could happen simply by calling the appropriate methods in the system that is going to be tested. The second phase is the actual execution or interaction with the SUT. In the third phase the result is evaluated and verified. The final destroys the fixture and put the SUT back into the state before fixture setup. Listing 2.1 shows a very simple unit test including each phase. Of course, it is possible to put the setup and teardown phase into separate methods (this would also decrease the number of duplicate test code). Separating a test into four phases allows the developer to know exactly which part of the system is being currently tested. (Mes07)

Each four-phase test is implemented by a test method executing each of the phases (see listing 2.1);

```
public void testFixtureInline()
                        throws Exception {
    // Fixture setup
    StationManagementFacade facade = new
       StationManagementFacade();
    BigDecimal stationId = facade.createTestStation("HBF")
       ;
    try {
        // Exercise system
        List trainsAtDestination1 =
                        facade.getFTrainssByOriginAirport(
                           stationId);
        // Verify outcome
        assertEquals( 0, trainssAtDestination1.size() );
    } finally {
        // Fixture teardown
        facade.removeStation( stationId );
    }
}
```

Listing 2.1: A simple four-phase test

2.5.2 Test Smells

(Mes07) defines a test smell as something that is a symptom of a problem. Because a test smell is a symptom, it does not describe where the problem might be originating from (as it could have multiple sources). There are two major kinds of test smells. *Code smells* that indicate a error inside the test code and *behavior*

smells, which could lead to a wrong outcome of a test during execution.

Most of the available literature focuses on how a test should be written. This section will give an insight into some error prone ways to design, write and refactor tests. At the same time, one or more possible solutions will be provided for each smell.

2.5.2.1 Code Smells

Code smells in the context of tests are recognized by developers when they maintain their test code. Code smells tend to affect cost of tests and they also are a sign that behavior smells will follow.

2.5.2.1.1 Obscure Test

As mentioned previously, tests act as documentation for the system being developed. On the other side, they are supposed to be a executable specification. This leads to a problem, because the granularity needed for both of the goals contradict each other. Tests that are too detailed or to little detailed could be hard to read and understand. A further problem could be that a test needs to much information to describe what is being tested. This symptom is called obscure test. Because obscure tests are heard to read, they increase the costs of maintaining test code and also can't act as documentation. The second issue with obscure tests is that the might lead to buggy tests and even hide errors in the production code. The main cause for obscure test might be a lack of motivation to regularly maintain test code, keep it clean and simple and the knowledge that the code written for tests is as important as the actual production code. Test code can be kept clear and simple by refactoring it regularly. As mentioned earlier, the

cause of obscure tests can be too much or too little information inside a single test. There are several other causes for a obscure test.(Mes07, p.186)

Mystery Guest A mystery guest is a test where the human reader is not able to see the purpose of a test because vital information is placed outside the test. The state of the SUT before running the actual test is the pre-condition for that test. Responsible for creating this state is the phase of Fixture Setup. The assertions made inside a test are responsible for verifying the post-condition of a SUT. A mystery guest depends on information that is located outside the test, making it difficult to understand the purpose of the test. When a test is difficult to understand without having to look up information outside the test, then it misses its purpose as documentation. An example for a mystery guest would be test that relies on some external resources (for example a csv file or the contents of a database) for fixture setup or for result verification. Changing the external resource could to lead unpredictable test behavior. A possible solution to mystery guests could be creating a fixture freshly for each test by an in-line setup (this would include creating the information which then acts as the contents of a file or a database within the fixture instead of reading them from an external resource). To avoid putting too much details inside the fixture, it is possible to use creation methods (Mes07, p.186-188).

Eager Test An eager test tries to validate and verify too much functionality inside a single test. It could be very difficult to tell apart the fixture from the actual testing. This behavior might be acceptable when the tests are executed step by step by an human who can decide when to abort a test. A solution to this problem could be writing single condition tests (Mes07, p.185).

General Fixture When trying to write a fixture that ‘fits all‘, it is possible to end

up with a fixture is too large for the currently executed test. This smell not only leads to difficult to understand relationship between the test and the fixture but also affects the performance of test automation. Large fixtures also tend to not be usable as documentation anymore. A possible solution would be to write an minimalistic fixture which can be used and extended by the tests (Mes07, p.190-192).

Irrelevant Information When setting up the fixture in-line a test, it is important to keep the complexity of the fixture in mind. Very long and complex fixture logic tends to obscure the pre-conditions of a test. The same goes for the verification logic of a test. The cause for this smell could be hard coded test data or a general fixture which then needs to be customized. Once more, the impact of irrelevant information might be hard to read test which then can't act as documentation (Mes07, p.190-192).

2.5.2.1.2 Conditional Test Logic

A sign for test logic that is too complicated is the presence of test code that might never be executed and is therefore considered a code smell. Conditional test logic tends to make tests more complicated than they actually are. Symptoms of this smell are loops and if statements inside a test. The extra complexity introduced by conditional test logic could lead to the project level smell of high maintenance cost. When using conditional test logic, one can't be as confidence about the outcome as with single path executions. Another issue with multiple paths of execution is lack of complete deterministic behavior each time the test is executed. It can also lead to tests that are verifying the wrong behavior and it becomes more and more difficult to catch bugs the more branches and loops the test contains. The reason for the existence of conditional test logic might be the missing ability of the SUT to deliver valid data for verification when the

test is executed. So the developers try to steer away the execution of the SUT from those parts. Another reason for the existence of conditional test logic is the verification of collections of objects (thus the loops) or the verification of complex objects. Conditional test logic leads to obscure tests. (Mes07, p.200)

Conditional test logic might also be the result of (Mes07, p.200ff):

Flexible Tests A test tries to verify parts of the SUT depending on under which circumstances it is run. This factor the execution path depends on could be anything from the time of the day to a specific value of a variable. The origin of this smell is the fact that it not always possible to make the SUT independent from its surroundings. A possible solution for this smell could be writing separate tests for each case.

Conditional Verification Logic This is very similar to flexible tests. But in this case not the execution path of the test depends on some kind of variable but instead the verification. For example, the state of a field is only verified when it met a previous condition.

Complex Teardown When a test is finished it needs to clean up after itself. Otherwise the next test could find the SUT in a corrupted state (especially when a common fixture is used). If a test uses multiple resources, each resource needs to be put back into a valid state. This can become very complex. The smell of complex teardown can be avoided by for example using fresh fixtures for each test or a automated teardown.

Multiple Test Conditions Trying to verify results for a set of input values will create multiple test conditions inside a single test. An example for this smell would be assertions inside a loop, depending on the value of the loop counter. This might cause very difficult to locate failed assertions.

2.5.2.1.3 Hard-to-Test Code

An mentioned in 2.2.1 Test-Driven Development helps backing up changes and modifications simply by using tests to verify the changes made, thereby reducing risk and fear. The greater the percentage of test coverage is, the more confident the developer can be about the changes made. On the other side though, writing tests is also added effort to the development cycle. So the aim should be a code base which can be easily tested. Sometimes this can be very difficult because of the structure of the code or the project. Especially GUI-code, code for multi-threaded and highly coupled software is difficult to test during development time. There could also be parts of the system that should be developed test-driven but simply are not visible to the tests (for example classes with a private constructor can not be easily instantiated). These problems demonstrate that hard-to-test code can make it sometimes impossible to get the desired amount of test coverage for enabling fearless development. (Mes07, p.209) The following list gives an more detailed overview of the circumstances leading to hard-to-test code (Mes07, p.210-2012).

Highly Coupled Code Code is highly coupled when unit tests for a class can not be written without also creating and testing several other classes. So it is not possible to run the test in isolation which violates the principle of unit tests. Test-Driven Development is a way to fight a high degree of coupling because it helps improving the design step by step and backing up each design modification by passing tests. As a result of this approach the developer gets a far less coupled code.

Asynchronous Code A symptom of asynchronous code are classes that can not be testes directly because the unit to be tested would return to the caller immediately. The execution of the test has to be coordinated with the state

of the SUT which sometimes is impossible. These attributes lead to higher complexity not only during the test execution phase but also during the verification phase. Asynchronous Code can also lead to tests that need very long time to finish (or can even be non-deterministic). This will decrease the motivation of the developers to follow the test-driven paradigm because long running tests will increase the level of frustration. A solution to this smell is to seperate the the logic which needs to be developed and tested from the execution mechanism of the concurrent or asynchronous parts.

Untestable Test Code The occurrence of obscure tests and/or conditional test logic will sometimes make it unclear if the test is even correct leading to potential buggy tests and causing high costs during maintenance. Overly complicated tests could be simplified by moving parts of the body into separate methods.

2.5.2.1.4 Test Code Duplication

Test code duplication happens when a test or a part of a test is repeated several times implicitly or explicitly. Tests could, for example, have the same fixture or verify the same results again and again or even execute expensive parts of the SUT several times, such as complex network or database operations. This is a disadvantage not only in regards of maintainability (because each time when the logic needs to be changed in one test, it has to be changed in all duplicating tests too)but could also lead to long execution time which will again increase frustration levels among the developers causing them to write less tests. Even worse, when the design of the SUT changes (for example new method signatures), all the tests will fail at once. This example shows that test duplication could lead to fear during development time and as a result could keep developers from changing and improving the SUT because they would have to alter a lot of existing test

code. (Mes07, p.213)

Duplicated test code if often introduced by ‘cut-and-paste‘ reuse because it seems to be an easy method of saving time. This mindset could be caused by missing refactoring skills or time pressure. Test code duplication can be avoided by using so-called extraction methods. This means putting all of the shared code into a separate place. This also works for reoccurring fixture, teardown and verification code. Another reason for test duplication are developers who try to ‘reinvent the wheel‘ simply because they might not know that somebody else might have written a similar or identical test or parts of a test. Using extracted methods with good names will also minimize this smell. (Mes07, p.214f)

2.5.2.1.5 Test Logic in Production

In some cases, the SUT contains parts that are only there to enable testing. These parts could be used to get access to the internal state of the SUT which otherwise would not be possible. They might also be used to alter the state of the SUT during testing. This structure leads to parts of the system that behave completely different in the testing environment as in production. An example for misplaced test code are so called test hooks. They decide if the production logic or the path designed for testing should be executed. Listing 3.4 shows a code fragment leading to this code smell. A reason for the introduction of test code in the production environment is often the desire to make the behavior of the SUT more deterministic during testing situations. This could be for example some kind of database access. A possible solution to this problem might be provided by the strategy pattern to encapsulate code designed for testing situations. Another form of test logic in production code are methods only used strictly by tests. These extra methods will make the SUT more complex and harder to understand and might be caused by the application of Test-Driven Development. This problem

can be avoided for example by extending the class which needs to be developed or tested and putting the extra internal logic in a subclass. The final cause for test logic in production code might be a dependency the production code has on some parts of the test logic. A symptom of this smell might be the fact that that the product can't be built without including the referenced parts of the test code. The danger here is that test code might be executed in a non-deterministic way leading to unpredictable results. (Mes07, p.217-221)

```
if (testing) {
        return hardCodedCannedData;
} else { // the real logic ...
        return gatheredData;
}
```

Listing 2.2: Meszaros2007]A test hook in production code.(Mes07, p.217)

2.5.2.2 Behavior Smells

Behavior smells will be notices more likely than code smells because they lead to non-compiling test code or failing tests. A common pattern among behavior smells are tests that used to pass but suddenly fail when executed another time. This pattern is called a fragile test (Mes07, p.13). Obviously, this pattern could be in fact a desired behavior to tell the developers that the changes they introduced broke the SUT. But that's is not what is described by the word fragile test because they even break when the code stays the same and only the circumstances change.

2.5.2.2.1 Assertion Roulette

When inspecting the output of a failed test, it is very difficult or even impossible to tell exactly which assertion failed. It gets even more difficult when the failure can not be reproduced, making it impossible to tell if the failure originated from defective code or some environmental issues. Assertion roulette can be caused by:

Eager Tests see paragraph 2.5.2.1.1

Missing Assertion Message When executing tests it is important to create output so that the reader of the tests result will be able to find exactly those tests and assertions that caused the failure. Assertion messages are even more important when the same kind of assertion is made several times in a row (e.g. sequential equality assertions). A possible solution to this cause can be simply adding at least simple assertion messages to the tests (Mes07, p.226f).

2.5.2.2.2 Erratic Test

Tests which sometimes pass and sometimes don't are so called erratic tests. Their result depends on when, where or even by whom they are executed. As one can see, this behavior is not acceptable when working in a team using Test-Driven Development. Simply removing such a test is not a option as it would result in source code not backed up by tests, which is not helpful when trying to reduce fear during development phase. Leaving it unchanged will cause failing test in a non-deterministic way. Finding out why a test sometimes fails is a difficult task. One has to collect data especially about the environment and the state of the SUT for both cases, when it passes and when it fails. Figure 2.9 describes a simple

process to find the reason for an erratic test. (Mes07, p.228)

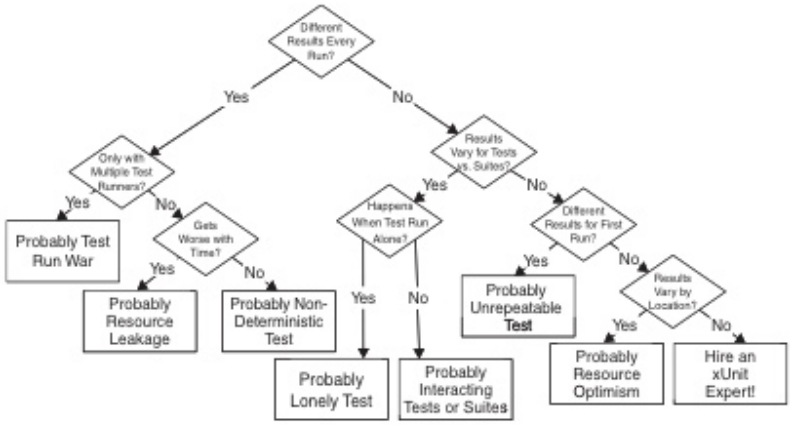

Figure 2.9: Troubleshooting an Erratic Test(Mes07, p.20).

There are several hints that point the developer to erratic tests. For example could some tests be depending on each other. They are called interacting tests. A test that used to pass and fails after a test is added to the suite or removed from it is a very specific hint for interacting tests. The same goes for tests that only fail if another test failed first. A common source for interaction between test is often a shared fixture. In a more general way, one could say that tests interact each time they share a common resource, for example a database. A possible solution for this dilemma would be using separate resources for each test, or at least provide a fresh fixture for each test. And if a resource has to be shared, it should be immutable or each test should leave the resource in the same state it found it in. (Mes07, p.231f)

Another cause for erratic test is resource leakage. A symptom for this smell are tests that keep getting slower or even fail. The reason for the decrease in speed if often a too greedy test or the usage of finite resources. Greedy in this context could also mean that needed resources are not freed after usage. A possible solution is a

guaranteed fixture teardown after the test has finished. Also, a test should never optimistically assume it would receive a resource when needed. Instead, the test should check if the resource is really available before continuing with its execution or even set up the resource during fixture phase. (Mes07, p.232f)

2.5.2.2.3 Frequent Debugging

When it is necessary to use manual debugging very often to find the cause for a failed test, (Mes07, p.248) suggests that it might be because of insufficient default localization. A test is supposed to tell where it failed and why. Not doing so might be a hint that a unit test is not detailed enough or there is a lack for an integration test, that would point out integration problems. Tests with too low granularity are often a reason of sloppy Test-Driven Development where unit tests were written for higher-level components but not for each method and for each line of code existing in the class. Another reason for this are often tests that are not run frequently enough, which is also against the Test-Driven Development philosophy. If the tests a run after each change, the error would show up much faster. Frequent debugging has a negative impact on the development process, as it slows down the development. As a consequence the productivity decreases and the results are far less predictable as they could be. A way to avoid frequent debugging is to really stick to the paradigms of Test-Driven Development. (Mes07, p.248f)

2.5.2.2.4 Slow Tests

Unit tests are supposed to run fast so that they are executed often (i.e. at each change) by the developers. So if a test runs very slow, it directly reduces the productivity of the developer executing the test in that this person is blocked by

running tests. A possible solution to speed up tests might me using a common fixture. But this approach may lead to other difficulties like erratic tests (see 2.5.2.2.2). Another solution might be using fake objects which run faster than the real ones. A way to find bottlenecks are profilers or just recording the start and end time of each test. (Mes07, p.250f)

Reasons for slow tests include:

Slow Component Usage A slow component is a component with high latency. A common example is the usage of a database. The average respond time for a database action like reading or writing takes up to 50 times longer than running the same operations against a in-memory data structure. So instead of using real databases for the unit tests, Meszaros suggests so called Test Doubles which run much faster. (Mes07, p.250f)

General Fixture Rebuilding the same (complex) fixture again and again by each test can lead to slow tests. In general, a fresh fixture is suggested because it reduces or removes interconnectivity between tests. So if it is absolutely necessary to use a shared fixture instead of a fresh general fixture, the shared fixture should be at least immutable to avoid unwanted side effects. But even with immutability, shared fixtures might lead to erratic tests. (Mes07, p.255)

Asynchronous Test Asynchronous Test sometimes take long to finish because they contain intended waiting periods to assure that some other process finished successfully. One single delay might not cause a long delay, but if added up, these tests could cause a significant increase in test duration. A way to avoid this problem is to test the asynchronous parts of the SUT synchronously. One could do that by extracting the logic inside the asynchronous parts into extra testable components. (Mes07, p.255f)

2.5.2.3 Project Smells

Project smells indicate that something is wrong on a project level. Still, they originate in one or more behavior or code smells. Project smells are a tool especially used by project managers to find faults in functionality, quality, usage of resources or project costs without having to write tests them self. (Mes07, p.13)

2.5.2.3.1 Buggy Tests

When applying Test-Driven Development, the tests reduce fear by providing assurance to the developers that the system works as required if all the tests pass. It is difficult though to prove that the tests themself are correct. A symptom for this smell are tests that fail not because of wrong production code but because the tests are wrong. Another symptom of buggy tests is production code that passes during development but fails in the production environment. False negative and false positive test can lead to unpredictable behavior. The main causes for buggy tests are fragile tests, obscure tests (see 2.5.2.1.1) or hard-to-test code (see 2.5.2.1.3). To avoid buggy tests, it must be assured that developers have enough time to learn to write good unit tests, refactor production code and test code and write the tests before writing production code. (Mes07, p.260ff)

2.5.2.3.2 Developers Not Writing Tests

Covering each line of production code by test code is an essential paradigm of the test-first approach. Not following this rule will lead to bugs that will be getting into the production environment. This behavior will cause the developed system to be in test debt and will lead to inefficiency and higher costs to add new functionality, remove bugs or maintain code. Finally, test debt will cause a fall

back to non-agile development. (Mes07, p.263)

The reasons for test debt often are:

Not Enough Time Developers not having enough time to write tests before writing production code often is a result of a too tight schedule or supervisors who don't understand the importance of the test-first approach.

Hard-to-Test Code see 2.5.2.1.3

To eliminate a project smell like this, is is necessary to have a person on the team with a greater view on the whole system. It is very difficult for a single developer to tell if each other member of the team has been writing tests before writing the production code. Most of the time they won't be able to tell if the whole source code base is even backed up by tests at all. If the reason for missing tests is a shortage of time, it is the duty of the managers to ensure that enough time is available for the developers to first write the tests, then the production code and refactor the created code to improve quality.

2.5.3 Test Patterns

Because the tests in Test-Driven Development act as documentation, they must be able to make it obvious to the reader, what exactly they are testing. Because they also are the specification of the SUT, it is of paramount importance, that they are easily to understand because only then they can be used to verify the proper behavior of the SUT. To do that, Meszaros suggests to always split them into four parts (Mes07, p.358), calling this pattern the Four-Phase Test (see 2.5):

- Fixture

- Exercise
- Verify
- Teardown

2.5.3.1 Fixture Setup Patterns

Two kinds of fixtures can be distinguished. The first one is the fresh fixture. This type of fixture creates its own SUT-state by assembling it for its private use, decreasing the odds for a erratic test and helping to see the tests as documentation. The idea is to use a fixture only once during the run of a single test. Tests must not rely on any assumptions made about the state of the SUT created by any other fixture. This approach will also avoid tests depending or interacting on each other. (Mes07, p.311f)

2.5.3.1.1 In-Line Setup

The first pattern for fresh fixtures is the In-Line Setup. That means that the constructors of the object are called directly by the tests in-line. It is important to put the fixture code at top of each single test, so they are more readable and support the documentation function of the test. Obviously, this pattern is most applicable when the test logic is not too complex. Otherwise it is better to extract the fixture code and put it into a own method. In-Line Setup can also be used when starting to write the tests for pieces of code that are yet to be developed. When the tests and the production code become more sophisticated, the fixture parts can still be executed. This would mean applying the steps of Test-Driven Development to the tests and not only to the actual code. (Mes07, p.408ff)

2.5.3.1.2 Delegated Setup

As already mentioned, parts of or the whole fixture can be extracted and put into separate methods. The fixture then is created by calling those methods. This pattern is called Delegated Setup. One of the principles of Test-Driven Development is to never duplicate code. When applying the In-Line Setup pattern, this principle can be easily violated at least for the test (leading to the same drawbacks one gets when having duplicate code in production code). The Delegated Setup pattern allows the developers to hand over the creation of objects to dedicated methods to avoid duplication and still allowing the tests to act as documentation. By moving the responsibility of object creation to dedicated methods, the smell of fragile tests can be minimized. (Mes07, p.411ff)

2.5.3.1.3 Prebuilt Fixture

The Prebuilt Fixture pattern is used for shared fixtures. Shared fixtures for tests are often used if a fresh fixture for each test would be too time- or resource consuming. By sharing the state of a system, tests can be executed significantly faster (Mes07, p.317). Although shared fixture often help speeding up tests, they also have some disadvantages. They can lead to dependencies and interactions between tests, rendering them useless as documentation and specification. They can lead to erratic, obscure and fragile tests because they will have to satisfy the requirements for all the tests they provide the fixture for (Mes07, p.318).

Applying the Prebuilt Fixture pattern means creating the fixture before executing the tests and sharing it among the tests. Example for such a fixture would be copying all the information needed from a database to the memory prior running the tests. (Mes07, p.429f)

2.5.3.1.4 Lazy Setup

Each tests is able to set up the fixture if the one it relies on is not ready. This pattern is called Lazy Setup. The initialisation still only happens once. All tests executed after the fixture creator will still use the shared fixture. A difficulty when applying the Lazy Setup pattern is to decide if the fixture can be torn down yet or not (maybe there are still tests left to run). (Mes07, p.435ff)

2.5.3.2 Result Verification Patterns

Result verification has the purpose to check the correct behaviour of the tested piece of code. Because of that, this phase of the test acts as specification for the system to be developed. Therefore the tests have to be self-checking. Self-checking means that the tests themself will verify the the outcome of the execution of the code to be tested. Only because of this feature it is possible to run Test-Driven Development cost-effective and to execute the tests frequently. When verifying, one has to distinguish between state verification and behavior verification. For state verification, some specific parts of the SUT are executed and afterwards the SUT's state if examined. Verifying behavior on the other side is more complex because of the dynamic nature of the SUT (some paths might be executed, others might not). Behavior is verified by observing outgoing method calls of the SUT. (Mes07, p.107ff)

This section will introduce state verification patterns since they are the primary way in Test-Driven Development to verify the correctness of the SUT.

2.5.3.2.1 State Verification

As already mentioned, self-checking means that tests have to verify the outcome of the execution of the tested code during their own execution. Therefore a test compares the final or the temporary state of the SUT after execution with a defined value. This pattern is called State Verification. State Verification is often used when developers are interested only in the outcome of a test execution, not how the SUT created that state. (Mes07, p.462f)

The state verification pattern comes in two variations. Those are, according to (Mes07, p.463f):

Procedural State Verification This variation expects the developer to simply use assert methods sequentially to verify that each part of a object or a system hast the expected value. Obviously, this simple approach also has the disadvantage that it could make a test obscure and thereby hard to read because complete verification might result in a large number of assertion-method calls.

Expected State Specification When applying Expected State Verification, one has to create and populate objects with the desired values. Using equality assertions, the received values from the SUT then are compared with the desired values. The pre-configured objects are called Expected State Specification. This approach will make the tests more readable. To increase readability, creation methods should be used to extract the object creation from the actual test. Of course, to perform result verification by using pre-configured objects, those types have to implement some kind of ‘equals‘-method which then is called by the unit testing framework.

2.5.3.3 Test Double Patterns

Often in Test-Driven Development, a developer has to test code, which depends on other parts of the system, which are not developed yet or simply unavailable. To achieve that, there exist several so called Test Double patterns, which allow the non-existing components to be replaced by test-specific equivalents. That equivalent is called test double. Obviously, the interface used for test doubles can later on be developed to productive parts of the whole system. During the fixture phase, a test double can easily replace the real (but unfinished) object. Developers should be aware that the production code later on will probably differ from the test double. (Mes07, p.462f) This is the point where the advantages of Test-Driven Development become visible again. Because the API stays the same and only the implementation changes, developers and managers can keep working on the unfinished parts of the system independently and without fear, because the will have the assurance that everything is working as expected, as long as the tests pass after removing the test doubles.

2.5.3.3.1 Mock Object

A Mock Object is a way to verify the behavior of a SUT. Because it is a variation of the Test Double patterns, it has to provide the same API as the real object will. Before usage, a mock object has to be configured by telling it which values it should return depending on the parameters passed on to it. The mock object can also be configured to check which method calls occurred and which did not, making it a strong tool for behavior verification. After creating and configuring the mock object, it hast to be installed into the SUT so it is used instead of the real object. It is a powerful tool to notice unwanted side effects like unwanted method calls. A disadvantage might be the fact, that the values for the

parameters handed over to the mock object have to be predicted and the actual values handed over during runtime have to meet those predictions. Otherwise the test will fail. Because mock objects need to be created and configured before the relevant parts ot a test are executed, they might decrease the readability of a test, turning it into a obscure test. (Mes07, p.544ff)

2.5.3.3.2 Fake Object

A Fake Object is an actual, but very simple implementation of the component that is needed for the test. One could say, that it might be the very early version of a component developed using Test-Driven Development. It might be also called a lightweight implementation of the component to be developed. Often, a fake object simply returns a pre-defined value regardless of what the input is. Fake objects are typically created during fixture. As with mock objects, fake objects too are used when the real implementation of a component is not available or would be too resource consuming to create and configure. The Fake Object pattern is not limited to objects and can also be applied to databases or any kind of web-service. (Mes07, p.551ff)

2.5.3.4 Test Organisation Patterns

As the system that is developed, grows, also the tests will grow. Ideally, no single line of code should be untested. To use the tests as documentation and specification, they need to be easily findable, readable and maintainable. There are generally two requirements for the organisation of tests: the tests available for the whole system or single components should be runable easily. At the same time tests belonging to similar parts of the system should be kept close to each other. This section will introduce a few patterns widely used to achieve these

goals. (Mes07, p.592)

2.5.3.4.1 Test Code Reuse Patterns

Patterns for test code reuse will ensure that test code is not duplicated. Test code duplication is inevitable as the amount and complexity of the tests and the system to develop rise. Duplication will lead to well known and severe problems already known from traditional object-oriented software development. The first pattern to fix test code duplication are Test Utility Methods. Each piece of logic that occurs more than once, it packed into subroutines which then can be called from any tests. Utility methods can be used for object creation, to retrieve objects from a shared fixture, for encapsulation of parts of the SUT which are no necessary for the current test and for custom assertions and verification. Test Utility Methods will always reduce the amount of code in a test. Sometimes this could make a test harder to read. Therefore, it is important to give the methods speaking names to ensure that the test can still be used as documentation. (Mes07, p.599ff)

Another variation for test code reuse is the Parameterized Test pattern. Tests often perform the same tests but with slightly different input. Normally, this fact will result will in a large amount of duplicated test code. To avoid duplication, the parts that are the same are extracted into a subroutine and can be called with different parameters. A parameterized differs from the previously mentioned utility method in that it will perform all four phases of a test (fixture, execution of the SUT, verification and tear-down). Again, this approach will help to avoid obscure tests. (Mes07, p.607f)

2.5.3.4.2 Test Class Structure Patterns

With a increasing number of tests and production components like classes, one must decide where to put the tests for each component of the production system. Because tests are documentation and specification at the same time, they must be organized in a way, so that they provide a big picture easily. The test organisation will also have a big impact on how the fixtures for the tests are designed and executed. A popular pattern for this issue is the Testcase Class per Class Pattern. A Testcase Class will contain all the tests for a single class of the SUT. This approach is usually the best choice when the development begins and the number of classes in the SUT is little. This pattern usually comes naturally when applying Test-Driven Development. (Mes07, p.617ff)

Another way to organise the tests is to group them by feature. This pattern is called Testcase Class per Feature. It is a more systematic approach than to just put all the tests for a class into a single testcase class. Instead, the developers have to decide which feature is verified by a test and group the tests depending on that decision. Grouping tests by the feature they verify, increases the specification role of tests in Test-Driven Development. A variation of this pattern is the Testcase Class per Method pattern, where tests are grouped based on the method they verify. This pattern is chosen often when a method needs to be tested with different parameters values. Another variation would be combining the tests depending on a user story. (Mes07, p.624ff)

2.6 Embedding Test-Driven Development Into a Appropriate Environment

Test-Driven Development is a practice highly embraced and introduced to a large audience in Kent Beck's 'Extreme Programming Explained' (Bec00). This section will show how Test-Driven Development is embedded into the XP methodology and benefits when surrounded by the principles and strategies defined by (Bec00).

2.6.1 Risk

On of the biggest challenges in the business of software development is the management of risk. Some examples denoted by (Bec00) and (Yli) are:

Schedule Slips Having to tell the stakeholders that the product won't be ready in time.

Project cancellation Project is cancelled before it was finished

Defect Rate The developed solution is never really used because of too much bugs in the final product.

Poorly Understood Requirements The solution developed does not solve the problems it was supposed to solve.

Changing Requirements Over time requirements can change. This could lead to a situation where the requirements identified at the beginning are no longer valid.

Missing Resources and Staff Turnover As agile development aims to create a

working and growing product each iteration of the development process, it might be difficult to have the required people and other resources available in time.

On a more abstract level, *schedule slips* can be addressed by shorter release cycles, limiting the scope of each slip. Each release should use one-to-four week iterations and within an iteration tasks should be scheduled for one-to-three days. *Project cancellation* can be avoided by asking the customer to choose those features for a release which make the biggest impact in term of business sense. This approach ensures that there is less to go wrong before going into production. Features chosen by the customer with the highest priority should be implemented first. When handling the *defect rate*, both perspectives of the stakeholders (customer and developers view) are considered by following the test-first approach and using acceptance tests written by the customer (see 2.2.2). *Poorly understood requirements* are one of the main reasons for failing software development projects (see 1). To handle this risk the customer should be a part of the core team at development site to continuously specify, clarify and also refine the requirements. As already mentioned, short release cycles should be preferred when applying Test-Driven Development or agile development in general. This leads to evolutionary changes during the development of a single new release in which the customer is welcomed to introduce new functionality as a substitute for not yet implemented functionality allowing to handle the risk of *changing requirements* and still keeping the amount of work predictable as new functionality is not simply added to the workload but substituted for a planned but not yet implemented functionality. The issue of *missing resources and staff turnover* is solved by giving the developers the responsibility for estimating the time they will need to complete a new functionality or feature. Giving them feedback about the time effectively taken enables them to improve future time estimates and avoid frustration caused by impossible to meet estimates. These actions, combined with the right communi-

cation culture and help for new team members decreases team fluctuation and leads to a higher job satisfaction. (Bec00)

2.6.2 Four Values

Personal goals have the disadvantage of often conflict with social goals which are defined with a longer time period in mind then shot-term personal goals. Therefore, a set of common values is required to give guidance to each member of a team when making decisions. (Bec00)

In XP and therefore Test-Driven Development, those values are:

- Communication
- Simplicity
- Feedback
- Courage

2.6.2.1 Communication

A lot of problems in software development can be traced back to missing or wrong communication, making it the first value of XP. Communication failures do not happen by accident but instead often are a result of circumstances that lead to bad communication. For example a programmer could be punished by the team leader of manager for mistakes made by that programmer and, as a result, keep bad news to him in the future. The aim is to establish the right amount of communication (i.e. keep the important information flowing) by introducing practices that won't

work without communication. One of those practices is Test-Driven Development as the created unite tests are not only a specification but also provide documentation for the piece of code they specify and thereby enable communication. Other communication supporting activities are pair programming and task estimation. (Bec00; AL12)

2.6.2.2 Simplicity

The first step of introducing and keeping simplicity to the project is to ask the developers, what the simplest thing might be, that would work in the particular situation. Simplicity demands a specific way of thinking. That is, the developer should not start wondering about the things that need to be implemented in future. XP follows the paradigm that it is better to implement the simplest solution available at the moment and to invest some extra effort if something more sophisticated is needed, than to implement a more advanced solution which may never be used. (Bec00)

The first value, communication, and simplicity are supposed to mutually support each other. The simpler a solution is, the less communication is needed. If a system is complex, communication helps in decreasing its complexity thereby making it simpler. (Bec00)

(MS10) list some of the benefits that come along with increased simplicity. These are:

- Better, more robust components.
- Better interoperation between components
- Better user comfort

- Better production processes
- Better maintenance and support

Test-Driven Development drives simplicity by forcing developers and customers to be clear about the software being implemented, as the first group has to write tests before starting to code, clarifying the interfaces and their usage. And the latter group is involved in specifying the requirements (e.g. by writing acceptance test).

2.6.2.3 Feedback

When talking about feedback, one has to consider the time scale it is given for. There is feedback that works at the scale of minutes and days. In terms of Test-Driven development and unit tests, feedback is given minute-by-minute as programmers create and run unit tests for each piece of change they establish, always having exact information about the state of the whole system. (Bec00)

The customer receives feedback through time estimations written by programmers for each new feature (i.e. story) they create. The whole team gets feedback from the instance that is responsible for tracking the completion state of tasks, allowing the team to see if they will likely finish the tasks in time. When looking at feedback from the point of view of acceptance tests, it works also for weeks and months, as customers write the acceptance tests or functional tests for all their use cases. As those tests fail or pass, the customer has a very detailed feedback about the current state of the system. If the current status conflicts with the planned schedule, adjustments can be made. (Bec00)

Following this guideline, it is possible to put a system into production at a very early stage, as the most important features are implemented first, giving develop-

ers and customers feedback about the decisions they made. Feedback supports the values of communication and simplicity. In the case of unit tests, feedback would be the results of the executed tests, making it simple to communicate about the quality of changes made or newly implemented. (Bec00)

2.6.2.4 Courage

Courage in the context of Test-Driven Development means making tough decisions even at a very advanced project state. Beck describes a case where a company found a architectural flaw that would cause the test score to decrease. Fixing some of the code would keep braking another part of the code. So the team decided to fix the underlying flaw in the architecture. This lead immediately to half the tests failing. But after some days of working, they were again heading toward completion. Another means of courage would be throwing code away that is not performing as expected or using development time to evaluate and implement design alternatives even if at the end only one design and its implementation stays in the project. As XP promotes a hill-climbing algorithm for design and development, it is likely to reach a local optima, where small changes would have only a small impact, but large changes would result in bigger benefits. This could mean throwing away large chunks of code. (Bec00)

2.7 Building the XP-Mesh Using Core Practices

The activities that define software development need to be structured to deliver good economic performance and quality. To achieve this, Beck introduces about a dozen practices and their relationships, which will be referenced as the XP-Mesh in this paper (figure 2.10).

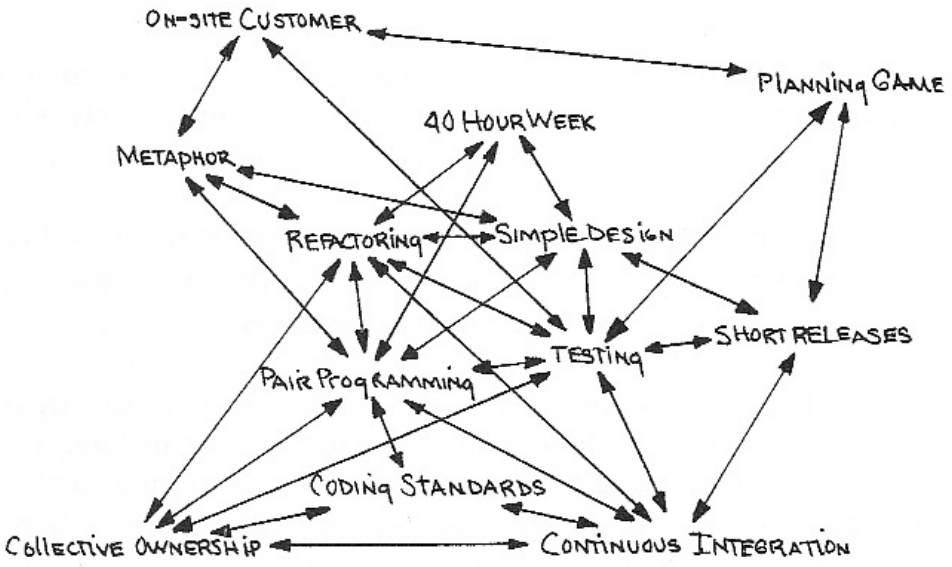

Figure 2.10: The XP-Mesh according to (Bec00, p. 70)

2.7.1 The Planning Game

Software development consist of two dimensions: the business aspects and the technical aspects. The aim is to find the right balance between those two dimensions, agreeing on an compromise about the possible and desirable(Bec00).

Components of the business aspect are (Bec00):

Scope Finding the right amount of functionality to implement during a specific amount of time, to be valuable in production.

Priority Decide which features need to be implemented first.

Composition of releases How much needs to be done so the software is a valuable addition to the business.

Dates of releases When should the software be presented or published.

Technical considerations contain (Bec00):

Estimates Try to estimate the time needed for each feature.

Consequences Provide information about the consequences that technical decisions have (e.g. in regards of productivity).

Process How exactly should the software be developed.

Detailed Scheduling Reduce overall risk of the whole project by implementing the most important segments first.

The technical considerations provide the data on which the business decisions are based on. The goal is to start with a real simple plan and refine it (just like the requirements) as the project evolves. A paramount part of the planning game is that the customers them self update the plan based on the estimates provided by the development team, providing them with a rough idea of the solution at the beginning. To do that, customers need to be at the development site, spotting potential misunderstandings, changes and opportunities. By planning and designing short releases, any mistakes would cause the product to be off-schedule for a few weeks or a few months at most. The most important metric for the success of the planning game is the ratio between calendar time and development time, providing a benchmark for feedback and communication. (Bec00)

The strategy is to plan what is needed in the concrete context. This could be for example the next release.

2.7.2 Short Releases

The paradigm to follow when applying Test-Driven Development should be to make the difference between each release as small as possible. Each release should contain only those features which provide the most value. The goal is to have a functioning release available as soon as possible. A prerequisite for short release intervals is the planning game to identify the most important stories, continuous integration for permanent available packaging, Test-Driven Development for a low defect rate and a simple design that is appropriate for this release. (Bec00)

2.7.3 Metaphor

The purpose of an metaphor is to help team members develop a common vocabulary and to be clear about the parts of a system and their relationship. An example for that would be if a pension calculating system would be seen as a spreadsheet. It is important to stick to the chosen metaphor in regards of identifying technical entities to guarantee consistency even at a advanced project state. As Beck points out, metaphor is similar to what could be seen as the architecture of a system, which is easy to understand and communicate. Metaphors are validated by feedback from the program code and the unit tests as they can tell if the metaphor is working for development. It is important to refine and improve the metaphor as development continues and to have a customer who supports thinking in terms of a metaphor. (Bec00)

2.7.4 Simple and incremental Design

(Bec00) sets up four basic rules to establish the ‘right‘ design of a software system of a component of a system. A design should: pass all tests (rule 1), must not contain duplicate logic (this also includes parallel class hierarchies) (rule 2), should make clear the intention of the developer (rule 3) and contain only the elements that are really necessary (rule 4). Because the most current design reflects the newest code status, simplicity of design is only possible when supported by other practices like refactoring, metaphors and pair programming.

Designing only what is needed at the moment helps fighting uncertainty because a feature that was already designed could vanish in the next release and also, if big chunks of a system change, the design could become inappropriate. (Bec00)

2.7.5 Refactoring

Refactoring means changing source code, the design and sometimes even the architecture of a system. The risk with refactoring is that it could possibly break the whole system, unless it is surrounded by other practices from the XP-Mesh such as (Bec00):

- Collective ownership, so anybody can make changes where they are needed.
- Simple design, that supports refactoring by its simplicity.
- Tests, so the developer will know exactly what is broken or is still working. This also reduces fear because if the tests pass, then the refactoring worked.
- Continuous integration, to make conflicts with other parts of the system visible in short time.

The aim of refactoring is to reduce complexity (which leads to simpler design), reduce or eliminate duplication or improve communication by increasing the readability and maintainability of the code.

2.7.6 Continuous Integration

In short, continuous integration means integrating new code and building the whole system several times a day and after completing each task (Bec00, p. 48).

Beck denotes, that no piece of code should stay unintegrated more than a few hours and after integration the code must pass 100 percent of tests. A big advantage of continuous integration is that it allows a single developer's code to be visible for everybody else in a short period of time. When something breaks, it is clear that the developer(s) who broke it, have to make the build pass again. A consequence of this approach might even result in throwing away a few hours worth of work and start from scratch. Risk is reduced significantly when applying continuous integration, because this practice will show collisions between different parts of code immediately during integration, making it possible to fix them early. (Bec00, p. 79f)

Continuous Integration also works this well because it is integrated into the XP-Mesh and tests can be run quickly to show conflicts and errors and programming is done in pairs (which reduces the overall potential of collisions) and refactoring allows to change code in smaller chunks, keeping the complexity small. (Bec00, p. 57)

2.7.7 Collective Ownership

Each member of a team has the right wo change any parts of a system. Without being guarded by tests, this fact would probably break the system eventually. Collective ownership will cause code that is too complicated to vanish over time, because everybody hast to understand and be able to talk about the code they are working on. So as soon as complicated code is found, it will get simplified, making the overall design less complex. If simplification leads to broken tests, parts of code might even be thrown away. (Bec00, p. 80f)

Collective ownership works best when combined with continuous integration to show and fix conflicts, tests to verify everything after changing it and coding standards to avoid unreadable code (Bec00, p. 56f).

2.7.8 Planning

Planning is a interaction between two participants. The economic side and the technical side. The technical side are the people who will build the system (and think about estimates, consequences and scheduling). Business is responsible for deciding what the purpose of the system should be (scope, priority, dates) (Bec00, p.72). The scope of the planning game is to roughly determine the scope of the upcoming iteration should be by balancing business priorities and technical estimates. In traditional software development, it would not be possible to start working with such a rough plan. But the XP-Mesh supports this approach in agile development by having an on-site customer who can update the plan if necessary and short releases so even big mistakes would cause a delay of a few weeks at most. (Bec00, p.54)

2.7.9 Coding Standards

Because of collective ownership, each developer is allowed to change any parts of the code. Without a coding standard it would be far more difficult to read code easily, decreasing its documentation and communication role. A coding standards is a set of rules and all developers need to write their code in accordance with those rules. (Bec00, p.53)

2.7.10 Pair Programming

The main rule of pair programming is that each line of code is written by two developers (using only one keyboard and mouse). The two people of the pair take different roles during the time they work together. The person controlling the input devices has to think about how to implement a specific feature. The second developer hast to act more strategically and consider if the development approach is suitable for the whole system, which test cases might be broken and if the created code can be simplified. The pairs are temporary. Usually they last for a few hours or a workday. (Bec00, p.51)

Pair Programming works because of the other factors of the XP-Mesh. Coding standards reduce discussions about coding style. Tests and a simple design ensure that both participants get a good understanding of the system and won't break the build. (Bec00, p.56)

2.7.11 On-Site Customer

The On-Site Customer is supposed to answer questions, solve disputes and make small-scale decisions.

3 Practical: Test-Driven Development of a VoIP Communication Client

For the practical part, a small prototype for a VoIP client had to be implemented for the Company BCT Electronics in Salzburg, Austria. Their core business is the development and manufacturing of communication terminals. It was established as a research and development subsidiary of the Commend group. The goal of the Project was to implement the VoIP client for the iOS platform to allow a user to connect to the terminals and control them.

The code should be developed using the Test-Driven Development approach. Unfortunately, the development could not be embedded in a real agile environment because the author was the only developer working on this project and also, there were not enough resources to implement a real agile approach.

This chapter will briefly demonstrate how the code was implemented by the practices described in the earlier chapters and sections. The main focus will be on how, starting with tests, Test-Driven Development lead to a object-oriented design with the help of refactorings.

3.1 Requirements

The following initial, very general requirements were provided:

Platform The system should work on iPads with supporting at least iOS version 4.0

ICX Protocol The ICX protocol is a serial protocol developed by Commend and used for the control of their communication terminals.

Video Streams It should be possible to display several video streams coming from surveillance cameras that are connected to the communication system over ICX.

IoIP Protocol For the transfer of audio information, the IoIP protocol had to be used. This is again an proprietary communication protocol developed and used by Commend. A implementation in C already existed and was integrated into the iOS VoIP client by an external contractor. The goal here was to allow the telephone components of the user interface to use this existing C implementation.

User Interface The user interface should consist of components which can be added and removed on demand. The components are: a telephone component to call other terminals and control hardware like door locks, parking gates and so on.

Transfer local camera signal It should be possible to stream the iPad's camera signal to any other communication terminal which supports the display of a video stream

Licence Management The user interface components should be accessible de-

pending on the licence available for the user.

3.2 Tools

This chapter will introduce two of the tools widely used under the iOS platform to develop software with the test-first approach.

3.2.1 OCUnit

OCUnit is the unit testing framework included in XCode, the IDE for Mac and iOS development. It was initially developed by Sen:te. Since it is integrated directly into XCode, a project supporting unit tests can be created simply by checking a check-box during project setup (Lee12, p. 35). All of the unit tests for the practical part of this thesis ware developed for OCUnit. To write a test case class, one only has to inherit from SenTestCase and import the proper headers. Assertions can be performed by simply using one the many STAssert*() macros provided by OCUnit. During runtime, OCUnit detects all subclasses and executes the tests (together with the setUp and tearDown methods). Test methods must not have a return value or parameters and must have the prefix 'test' in lowercase. A disadvantage is the missing feedback for the developer since there is no kind of green or red bar indicating if all tests passed or not. Instead the developer must have a look into the output created by OCUnit to check the results. (Lee12, p. 36ff)

To obtain information about code coverage, on has to simply link against a library called profile_rt (which is provided by apple) and enable the generation of coverage information in the project settings.

3.2.2 Continuous Integration with Hudson

Hudson, now knows as Jenkins, a platform independent tool for continuous integration (see 2.7.6). Jenkins can be configured to observe the source code repository for any changes and to perform several actions upon any change that happens on that repository. On of them is to execute a build job. To start Jenkins, simply load and start the jenkins.war file. Jenkins supports several version control systems such as cvs, subversion and git. Jenkins can start the xcode build process by calling the command line tool xcodebuild, which is provided by apple and designed to perform a customized build. It is possible to tell Jenkins what should be done if a build succeeds, fails or anything in between. (Lee12, p. 53ff)

3.3 Development Phase

As mentioned in 3.1, on of the requirements was to support different types of user interface components. The components could be a range of control or display elements and some of them even support interaction with the user (like performing some kind of command or be dragged around on the screen). The metaphor used for these components is Screen Elements. The screen elements will be represented by objects of the type UIElement. Each UIElement will be member of a so called screen. Multiple screens will be allowed and navigation will be accomplished by gestures. There will be a controller class called VoIPViewController which is assumed to be implemented for the scope of this paper. VoIPViewController is responsible for managing all the screen of the application and also recognising application wide gestures like multi-finger swipes, pinch and zoom. At least one screen is always available and new UIElement objects will be placed on the screen currently visible.

```
@interface UIelementTestClass : SenTestCase

@end

@implementation UIelementTestClass
- (void)testConstructEditWindow {
    VoIPViewController *viewController =
                                    [[VoIPViewController alloc
                                        ]
                        initWithNibName:
                        @"VoIPViewController" bundle:nil];
    GHAssertNotNil(viewController, @"WebView: %@",
            viewController);
    UIElement *simpleRect = [[UISimpleRectElem alloc]
    initWithFrame:CGRectMake(120, 120, 150, 200)
         andViewController:viewController];
    GHAssertNotNil(viewController, @"WebView: %@",
            viewController);
}
@end
```

Listing 3.1: The first test for a screen element

The first implementation is very simple and will obviously pass the test.

```
- (id)initWithFrame:(CGRect)frame
                    andViewController:(UIViewController*)
                        viewController
```

```
{
    self = [super initWithFrame:frame];
    if (self) {
        _viewController = viewController;
    }
    return self;
}
```

Listing 3.2: First implementation of a screen element

Almost each screen element will be a subclass of UIElement. What they all have in common is that the are UIViews and all are UIElements. This lead to the decision to keep the parts that are responsible for rendering in the base class and move the changing parts to the subclasses.

All screen elements need to react upon touch events. Touches on buttons will trigger some kind of action (like making a phone call) or will perform some kind of screen manipulation (like adding or removing of other UIElements). Other touches will cause the application to display of an overview of all screen or change the mode of the application to either work-mode or edit-mode. The messages sent to such touch strategy will be either that a touch started, ended or the finger was moved around the screen. What is already clear at this moment, is the fact that the only thing that changes for each UIElement object is the way it reacts on touches. So over time, TouchStrategy will become an interface (called protocol in Objective-C) with several implementations. Each UIElement will have a field of type TouchStrategy. As the name already tells, this approach will implement the Strategy pattern. This pattern was not obvious in the beginning, but became clear after implementing two different touch algorithms for UIElements. The following listing is a simplified version of the tests written for the initial TouchStrategy

class.

```
@interface TouchStrategyTestClass : SenTestCase

@end

@implementation TouchStrategyTestClass
- (void)testTouchBegan {
        UIElemen element = UIElement *simpleRect =
                        [[UISimpleRectElem alloc]
                initWithFrame:CGRectMake(120, 120, 150,
                        200)
    TouchStratedy *touchStrategy = [[TouchStratedy alloc]
            init];
    GHAssertNotNil(touchStrategy, @"Touch: %@",
            touchStrategy);
    [touchStrategy touchesBegan: touches withEvent: event
            forElement:
    simpleRect];
    GHAssertTrue([simpleRect touched], @"");

- (void)testTouchMoved {
    TouchStratedy *touchStrategy = [[TouchStratedy alloc]
            init];
    GHAssertNotNil(touchStrategy, @"Touch: %@",
            touchStrategy);
    [touchStrategy touchesMoved: touches withEvent: event
            forElement:
```

```
    simpleRect];
    GHAssertTrue([simpleRect moving], @"");
}

- (void)testTouchEnded {
    TouchStratedy *touchStrategy = [[TouchStratedy alloc]
        init];
    GHAssertNotNil(touchStrategy, @"Touch: %@",
        touchStrategy);
    [touchStrategy touchesEnded: touches withEvent: event
        forElement:
    simpleRect];
    GHAssertFalse([simpleRect touched], @"");
}
@end
```

Listing 3.3: Initial test for TouchStrategy

The implementation belonging to this tests simply set a field. The base class is only about receiving the actual message, not what happens inside. Because it is clear that UIElement is going to need its individual touch strategy, another strategy class had to be created. In this case, it will be the touch strategy for a delete button that removes the component it belongs to from the screen and from the application. The element should be removed from the screen after the touch ended. As always, the test are created first.

```
@interface TouchStratedyDeleteButton : SenTestCase

@end

@implementation TouchStratedyDeleteButton

        [...]

- (void)testTouchEnded {
        UIElement *simpleRect =
                        [[UISimpleRectElem alloc] init];
    TouchStratedyDeleteButton *touchStrategy = [[
        TouchStrategyDelete alloc]
    init];
    [simpleRect setTouchStrategy: touchStrategy]
    GHAssertNotNil(touchStrategy, @"Touch: %@",
        touchStrategy);
    [...]
    [touchStrategy touchesEnded: touches withEvent: event
        forElement:
    simpleRect];
    GHAssertFalse([simpleRect touched], @"");
    //check if element still present
    boolean present = [[[[VoIPViewController instance]
        currentScreen]
    elementsCurrent] containsObject:simpleRect];
    GHAssertFalse(present, @"");
}
```

```
@end

//actual implementation of touchesEnded

- (void) touchesEnded:(NSSet *)touches
                withEvent:(UIEvent *)event forElement:(
                      UIElement*) element {
    UICommonOverlayView *overlay =
                (UICommonOverlayView*)[element superview];
    NSLog(@"Deleting");
    UIElement* elemToDelete = [overlay underlyingElement];
    CAAnimation *animation = [Init getFlipAnimation];
    [elemToDelete removeFromSuperview];
    [[element superview] removeFromSuperview];

    [[[[VoIPViewController instance] currentScreen]
          elementsCurrent]
                        removeObject:elemToDelete];
    [[VoIPViewController instance] storeProfile];
    [[[element superview] superview]
                addAnimation:animation forKey:@"
                      toogleFromScreen"];
}
```

Listing 3.4: Creating another strategy to handle touches

At this point a developer can already sense that there would be a lot of duplication. The next step was to extract the parts that stay the same and put the changing parts into separate implementations. Therefore, TouchStrategy was turned into

a interface containing the declaration of the three methods needed. A reference to a TouchStrategy is held by each UIElement. This refactoring will be backed up by the already existing tests, with only little changes needed.

These snippets are shortened versions of the tests and the production code implementation to demonstrate how the process of refactoring and designing works with Test-Driven Development.

4 Concluding Remarks and Future Work

This thesis examined the different aspects of Test-Driven Development and how they have to be combined to deliver a satisfying end product to the customer. The first chapter covered how traditional development processes approach software development and why they often fail. The main reasons is the focus on formal guidelines and inflexibility towards new changes in the specification and requirements. Because of the missing possibility to go back to the requirement analysis phase in a lot of sequential development processes, the development team will run into big problems if the requirements are not fully understood before the coding phase begins. Even partly iterative processes will run into this problem because they often provide iterative behavior only for the phases after requirements analysis.

The second chapter demonstrates the aspects of Test-Driven Development. It begins by addressing the topic of agile development and examines the roles of the members of a agile team. The role of fear during the development process is examined and the focus is laid on why it blocks successful development. Then, the thesis goes on to describe how Test-Driven Development tackles this issue. The next section looks into a traditional Test-Driven Development cycle and works out the role of refactoring and how it helps to achieve object-oriented, coherent

and simple deign. The influence of Test-Driven Development is discussed in the next chapter by referencing two papers which examined a lot of different studies and conducted their own studies. These thesis comes to the conclusion that Test-Driven development leads to significant simpler, more coherent code and also decreases the error rate, which is a important external quality indicator.

It is of paramount importance to organise and write tests in a way they can act as documentation and specification. There are different kinds of ‘smells‘ that should be avoided. Each smell can be on the level of the project, the behavior of the system or the code itself. Code smells indicate that tests are erroneous, obscure or too complicated to read. They make it hard for the tests to act as documentation. The smells on higher levels can lead to tests being ignored or not written. These smells can cause severe problems in the actual production system.

The final part of the section about the organisation of tests introduces some of the widely used test pattern in modern Test-Driven Development. These patterns are categorized into fixture patterns, verification patterns and test organisation patterns. Fixture patterns describe different ways to create the context for a test. Test Double patterns allow the developer to write tests for parts of the system that are either no developed yet or are not accessible for the testing environment. Test Organisation pattern describe how to structure tests classes and how to write test code that can be reused.

The last section of the second chapter discusses the right environment to get the most out of TestDriven Development. Beginning with the role of fear in software development it then continues to give insight to the four values of agile development and how to build the XP-Mesh using twelve core practices. Without these practices surrounding Test-Driven Development it is hardly possible to harvest the full potential of it.

Finally, the last chapter is a short overview of the implementation phase of a VoIP client for the iOS platform that was developed using the test-driven approach. Some of the used tools are introduced and it is shown how, through abstraction and refactoring, it is possible to obtain object-oriented, simple design. For that, the thesis describes the development process for the user interface components and the logic that handles touch related events like gestures and multi-finger touches.

4.1 Future Work

Although Test-Driven Development has been around for several years, there are still problems that are a challenge to address with this kind of development practice. An area of future research will be making tests independent from factors that are not controllable by the developer.

It is also difficult to develop graphical user interfaces purely with the test-first approach. although there has been a lot of improvement in the last few years. Finally, it is still a problem to write test for software that is going to support asynchronus calls and/or is going to be heavily multi-threaded.

Bibliography

[Adz11] Adzic, G.: *Specification By Example*. Hanning, 2011

[AL12] Ambler, Scott ; Lines, Mark: *Disciplined Agile Delivery*. IBM Press, 2012

[Bec00] Beck, K.: *Extreme Programming Explained*. 2nd. Addison-Wesley, 2000

[Bec02] Beck, K.: *Test Driven Development: By Example*. Addison-Wesley Professional, 2002

[BFC09] Beck, Kent ; Fowler, Martin ; Cockburn, Alistair: *Principles behind the Agile Manifesto*. http://http://agilemanifesto.org/principles.html. Version: 2009

[Boe88] Boehm, Barry W.: A Spiral Model of Software Development and Enhancement. In: *IEEE Computer* 21 (1988), Nr. 5, 61-72. http://weblog.erenkrantz.com/~jerenk/phase-ii/Boe88.pdf

[DH04] Dubinsky, Yael ; Hazzan, Orit: Roles in Agile Software Development Teams. In: Eckstein, Jutta (Hrsg.) ; Baumeister, Hubert (Hrsg.): *Extreme Programming and Agile Processes in Software Engineering,*

5th International Conference, XP 2004, Garmisch-Partenkirchen, Germany, June 6-10, 2004, Proceedings Bd. 3092, Springer, 2004 (Lecture Notes in Computer Science). – ISBN 3–540–22137–9, S. 157–165

[EMT05] Erdogmus, Hakan ; Morisio, Maurizio ; Torchiano, Marco: On the Effectiveness of the Test-First Approach to Programming. In: *IEEE Trans. Software Eng.* 31 (2005), Nr. 3, S. 226–237. http://dx.doi.org/http://dx.doi.org/10.1109/TSE.2005.37. – DOI http://dx.doi.org/10.1109/TSE.2005.37

[Fin06] Fink, Kerstin ; Ploder, Christian (Hrsg.): *Wirtschaftsinformatik als Schlüssel zum Unternehmenserfolg*. Deutscher Universitäts-Verlag, 2006 http://books.google.at/books?id=ChvaewMMmiYC&pg=PA90&dq=heavyweight+software+model&hl=de&sa=X&ei=UAxPUf2-Dc3DswaZuoDgBA&redir_esc=y#v=onepage&q=heavyweight%20software%20model&f=false

[FP10] Freeman, S. ; Pryce, N.: *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley, 2010 (Addison-Wesley Signature Series)

[Gus02] Gustafson, David: *Schaum's Outline of Theory and Problems of Software Engineering*. 2002 (Schaum's Outline Series)

[HD08] Hazzan, Orit ; Dubinsky, Yael: *Agile Software Engineering*. Springer, 2008

[JM07] Jeffries, Ron ; Melnik, G.: Guest Editors' Introduction: TDD–The Art of Fearless Programming. In: *Software, IEEE* 24 (2007), Nr. 3, S. 24–30. http://dx.doi.org/10.1109/MS.2007.75. – DOI 10.1109/MS.2007.75. – ISSN 0740–7459

[JS06] Janzen, David S. ; Saiedian, Hossein: On the influence of test-driven development on software design. In: *In Nineteenth Conference on Software Engineering Education & Training*, 2006, S. 141–148

[JS08] Janzen, David ; Saiedian, Hossein: Does Test-Driven Development Really Improve Software Design Quality? In: *IEEE Softw.* 25 (2008), März, Nr. 2, 77–84. http://dx.doi.org/10.1109/MS.2008.34. – DOI 10.1109/MS.2008.34. – ISSN 0740–7459

[Lee12] Lee, G.: *Test-Driven IOS Development*. Addison Wesley Professional, 2012 (Developer's Library). http://books.google.at/books?id=Ji5OYAAACAAJ. – ISBN 9780321774187

[Mes07] Meszaros, G.: *xUnit Test Patterns - Refactoring Test Code*. Addison-Wesley, 2007 (The Addison-Wesley Signature Series)

[MS10] Margaria, Tiziana ; Steffen, Bernhard: Simplicity as a Driver for Agile Innovation. In: *IEEE Computer* 43 (2010), Nr. 6, S. 90–92. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/MC.2010.177. – DOI http://doi.ieeecomputersociety.org/10.1109/MC.2010.177

[PWB09] Petersen, Kai ; Wohlin, Claes ; Baca, Dejan: The Waterfall Model in Large-Scale Development. In: Bomarius, Frank (Hrsg.) ; Oivo, Markku (Hrsg.) ; Jaring, Päivi (Hrsg.) ; Abrahamsson, Pekka (Hrsg.): *Product-Focused Software Process Improvement, 10th International Conference, PROFES 2009, Oulu, Finland, June 15-17, 2009. Proceedings* Bd. 32, Springer, 2009 (Lecture Notes in Business Information Processing). – ISBN 978–3–642–02151–0, 386-400

[Rog04] Rogers, R. O.: Acceptance Testing vs. Unit Testing: A Developer s Perspective. In: Zannier, Carmen (Hrsg.) ; Erdogmus, Hakan

(Hrsg.) ; Lindstrom, Lowell (Hrsg.): *Extreme Programming and Agile Methods - XP/Agile Universe 2004, 4th Conference on Extreme Programming and Agile Methods, Calgary, Canada, August 15-18, 2004, Proceedings* Bd. 3134, Springer, 2004 (Lecture Notes in Computer Science). – ISBN 3–540–22839–X, S. 22–31

[Sca01] Scacchi, W.: Process Models in Software Engineering. Version: 2nd, 2001. http://www.ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SE-Encyc.pdf. In: *Encyclopedia of Software Engineering*. 2nd. John Wiley and Sons, 2001

[Som11] Sommerville, Ian: *Software Engineering*. 9th. Addison-Wesley, 2011

[Yli] Ylimannela, Ville: A MODEL FOR RISK MANAGEMENT IN AGILE SOFTWARE DEVELOPMENT.