WWW.JAMK.FI

AUTOMATED TESTING PERFORMED BY DEVELOPERS

Tuukka Turto

Master's Thesis

5 2013

Master's Degree Programme in Information Technology

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Author(s)
TURTO, Tuukka

Type of Publication
Master's Thesis

Date
5.5.2013

Pages
102

Language
English

Confidential

( ) Until

Permission for web publication
(X)

Title
AUTOMATED TESTING PERFORMED BY DEVELOPERS

Degree Programme
Master's Degree Programme in Information Technology

Tutor(s)
RANTALA, Maj-Lis
SALMIKANGAS, Esa
RINTAMÄKI, Marko

Assigned by
Digia

Abstract
The commissioner of the thesis was Digia Plc and the target of the thesis was to research and improve automated testing performed by the software developers. The main topics of the thesis were research, development and training. Various technologies were evaluated in order to find good set of tools to support the teams. Trainings were arranged related to these technologies and tools for the teams. In addition to that, there were two surveys that were used to evaluate how the software developers felt about automated testing.

A great deal of attention was given to various problems and challenges that could hinder testing. Some of the teams were more active in using automated testing; however, in general the developers felt that the automated testing makes sense and helps them in their daily work. Different teams had slightly different focus on their testing effort, depending on the needs of the team.

It was observed that introducing automated testing into a legacy application is not an easy task and it might require some unconventional design choices. The tests also require attention and maintenance as the system evolves and changes.

During the research an improvement in perceived quality of the software was observed. The developers gained a better understanding how the components of the system work together and had less defects in their code. The difference of regression rate between the developers also decreased.

Keywords
Automated testing, continuous integration, software quality, action research

Miscellaneous

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖN
KUVAILULEHTI

Tekijä(t)	Julkaisun laji	Päivämäärä
TURTO, Tuukka	Opinnäytetyö, ylempi ammattikorkeakoulututkinto	5.5.2013
	Sivumäärä 102	Kieli Englanti
	Luottamuksellisuus ( ) saakka	Verkkojulkaisulupa myönnetty (X)

Työn nimi

AUTOMATED TESTING PERFORMED BY DEVELOPERS

Koulutusohjelma

Master's Degree Programme in Information Technology

Työn ohjaaja(t)

RANTALA, Maj-Lis

SALMIKANGAS, Esa

RINTAMÄKI, Marko

Toimeksiantaja(t)

Digia

Tiivistelmä

Opinnäytetyö tehtiin Digia Oyj:lle ja sen tarkoituksena oli kehittää ohjelmistokehittäjien suorittamaa automaattitestausta. Erilaisiin tekniikoihin ja teknologioihin paneuduttiin kattavasti ja niitä vertailtiin. Vertailun perusteella valittiin yhteisesti käytössä olevat työkalut. Testauksen eri painopistealueisiin valikoitui joukko tekniikoita, joiden käyttöönottoon järjestettiin koulutusta. Lisäksi toteutettiin kaksi kyselyä, joilla kartoitettiin ohjelmistokehittäjien mielipiteitä liittyen automaattiseen testaukseen ja sen hyödyllisyyteen.

Työssä paneuduttiin erityisesti ratkaisemaan testausta estäviä ongelmia ja esitettiin erilaisia ratkaisumalleja niihin. Osa kehitykseen osallistuneista tiimeistä ottivat automaattisen testauksen aktiiviseen käyttöön. Yleisesti ottaen, kehittäjät kokivat automaattisen testauksen mielekkääksi ja työtä helpottavaksi. Eri tiimeissä testauksen painopiste muotoutui omanlaisekseen, tiimin sen hetkisisten tarpeiden mukaan.

Samalla huomattiin, ettei automaattisten testien tuominen vanhaan järjestelmään ole helppo toimenpide ja se saattaa vaatia totutusta poikkeavia suunnitteluratkaisuja. Testit myös vaativat jatkuvaa ylläpitoa järjestelmän muuttuessa.

Tutkimuksen aikana havaittiin järjestelmän subjektiivisesti havannoidun laadun parantuneen. Kehittäjät saivan paremman kokonaiskuvan järjestelmän komponenttien toiminnasta ja heidän koodissaan oli vähemmän virheitä.

Avainsanat (asiasanat)

Automaatiotestaus, jatkuva integraatio, ohjelmiston laatu, toimintatutkimus

Muut tiedot

Contents

List of Figures

List of Tables

Listings

1 Introduction

1.1 Commissioner

The commissioner of the thesis was Digia Plc, which is a Finnish software solutions and services company. Digia delivers ICT solutions and services to various industries, focusing especially on finance, public sector, trade and services and telecommunications. Digia operates in Finland, Russia, China, Sweden, Norway, Germany and in the U.S. The company is listed on the NASDAQ OMX Helsinki exchange (DIG1V). (Digia, 2012.)

1.2 Objective of Thesis

Very complex software systems are slow to test manually and there is pressure to shorten the time that is needed for testing before releasing the software. At the same time more companies are moving to agile methodologies where the old *"Testing is responsibility of testers"* is at least partly replaced with *"Testing is everyone's responsibility"*. Automation is used in order to speed up the test execution and to ensure that the tests are executed without mistakes.

The objective of the thesis is to evaluate different ways of performing automated testing in a software development company, map out some of the most common pitfalls and offer possible solutions to them. The focus is on the testing of legacy code and introducing new technologies and methodologies to support this endeavour. This is carried out in order to improve both internal and external quality of the software systems and improving working conditions of the software developers. Unit, integration and end to end testing are covered in the thesis. A literature review and an action research are used as research methods.

Automated testing was taken into use as everyday part of work. The software developers are responsible of writing, executing and maintaining automated tests that are used to ensure that the software works as intended. Before and after the large scale rollout of automated testing, a survey was executed in order to gauge how the developers view the automated testing and how it affects to their views about the software system.

The area of automated testing is huge and the present thesis can cover only a tiny scratch of it. The thesis does not explore for example automated user interface testing, performance testing or security testing. It also does not have enough space to cover parallel execution of tests and automated test environment management. Chapter 17.3 outlines some of the most interesting subjects that could not be covered and that could be researched later to build on top of the research done in the present thesis.

1.3 Outline of Thesis

The first part, consisting of chapters from 2 to 13, is a literary review, which forms the theoretical foundation for the thesis.

The second part, chapter 14, presents how the theory presented in the first part was put into use by the commissioner.

The final part, chapters from 15 to 17, is used to wrap up and present the results of the thesis.

The graph in Figure 1 shows the main concepts covered in the thesis and how they relate to each other. It also serves as a graphical index, which can be used to quickly locate some of the main parts of the thesis.

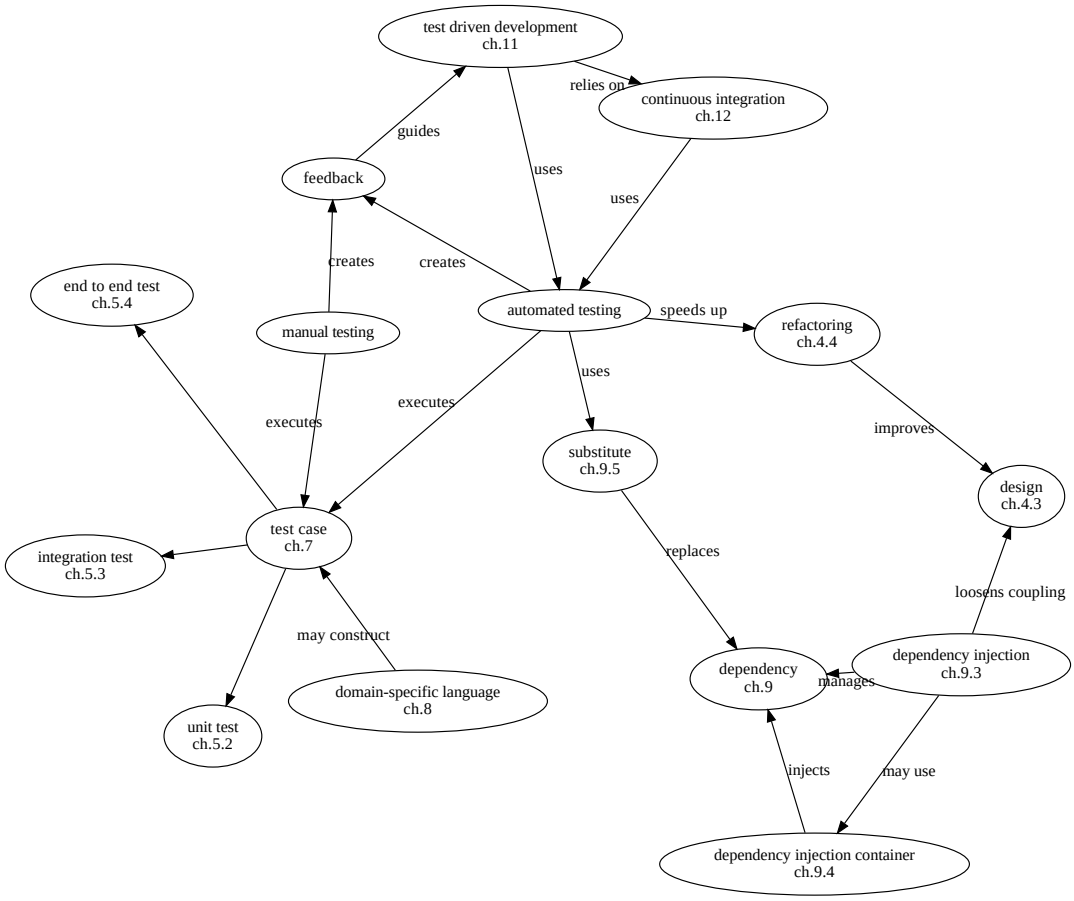

Figure 1: Overview of the thesis

2 Testing

2.1 Definition of Testing

Myers, Sandler and Badgett (2004, 6) define testing simply as a process of executing a program with the intent of finding errors.

Loveland, Shannon and Miller (2004, 6) narrows the scope down quite a bit by stating that the goal is to find the defects that matter, instead of exhaustively trying to find each and every one. In a large-scale software, finding all the defects is not even possible.

In his master's thesis Pohjolainen (2003, 9) lists many different definitions that have a slightly different focus or scope. Almost all of them have certain common elements, which are listed below as follows:

systematical

Only with systematical testing is it possible to repeat the testing process time after time. If the testing is neither planned nor structured, it can be impossible to compare the results from two different testing periods.

test material

Often testing requires test material that is used to simulate various inputs to the system. At the simplest, these are just lists or tables of values that a tester manually inputs to the system. A more complex material might consist of multiple documents laid out in a very specific manner that are automatically processed by the system under test.

specifications

The specifications of the system are essential, because it is nearly impossible to test a system without a clear understanding of how it is supposed to work.

evaluation

Tests are used to evaluate the system under test in a way or another. The result of the evaluation can be as simple as *"Runs fine, doesn't crash under load"* or as complex as a list of tested components and all defined special cases they were not able to handle. The point is that testing produces results and those results need to be evaluated. Based on the evaluation, actions might or might not be taken.

In the present thesis, testing is treated as an act of finding out if a system under test is working correctly in a given case.

2.2 Anatomy of a Good Test

Quality of the tests is directly related to quality of testing in general. Writing automated tests is not that difficult, but writing good automated tests can be rather hard. In fact, if the tests are not of good quality, they might cause more harm than good. This is because they might be hard to maintain or they might be testing wrong things and giving incorrect results. Surveys done by Hutcheson (2003, 3) found out that the test automation is the most difficult test technique to implement and maintain.

A good test is focused to a specific part of the software. It tests that specific functionality and nothing else. Results of the tests should be clear and quantifiable. The test should be repeatable, so the results can be verified by repeating the test. Repeatable tests can be used to gauge the maturity and quality of the software.

Because writing a large scale software is a group effort, the tests that are used to test that software should be readily understandable by the group. Often somebody else than the original author of the test has to maintain and change it. The tests should therefore be terse, clear and understandable. Preferrably they should have no conditional logic inside them at all.

2.3 Summary

There are many definitions for testing and application of tests. Because of this it might be hard for people to communicate their intentions clearly, unless common language is established. When the common language has been defined, it is easier to focus on the details and write good tests that are helpful for the team.

The test code should be treated as equally important as the production code. This means that they have to be written well, maintained and improved as the time passes. The test code usually does not get shipped to the customer; however, it is used to verify the correctness of the production code. By neglecting the test code the team would be indirectly neglecting the production code.

3 Motivation for Software Testing

3.1 Measuring Quality

Testing alone does not improve the quality of a software system. It can be used to measure the quality of a system or verify that the quality is on a certain level, however it alone can not improve the quality. After the code has been written, testing will not change how it behaves. Whittaker describes that Google has an approach where they stopped treating testing and development as separate disciplines. The developers own the quality of the software and are tasked to do both the testing and the development so close to each other that they become indistinguishable from each other. (Whittaker 2011.)

Tests can be used to gauge the quality of the system under test. With sufficiently large amount of tests it is possible to analyze which components are most likely to have unknown defects and would therefore require more testing. The execution time of tests on the other hand shows how the performance of the system has changed since the tests were ran last time.

3.2 Reducing Costly Errors

Testing can be used to reduce costly errors. In some domains, operating a faulty software can have devastating effects. Such industries include aerospace, nuclear and medical industries. These industries have very high requirements for traceability and delivering as error free software as possible, yet mistakes still happen. For example, NASA lost Mars Climate Orbiter in September 23, 1999 because of a software error (Stephenson, Mulville, Bauer, Dukeman, Norvig, LaPiana, Rutledge, Folta and Sackheim, 1999, 6). There were other contributing factors, but the root cause was *"failure to use metric units in the coding of a ground software file, Small Forces, used in trajectory models"* (Stephenson et al., 1999, 7). During the investigation it was concluded that end-to-end testing to validate that the software in question was working correctly and according to the specifications did not appear to be accomplished (Stephenson et al., 1999, 24).

Another example of costly error is the software glitch that caused initial loss of 440 million dollars to Knight Capital. Essentially, a new trading algorithm was being tested and it traded shares at loss at very high frequency. (Olds 2012.) The software error caused abnormal trading that in turn effected to the prices of the traded stocks.

Both of these errors had a very high financial impact and had a negative effect on the image of the respective companies. A more strict test and review process might have caught these errors. It is interesting to note that in the case of Knight Capital the software was actually being tested when the error occurred and caused the abnormal trading.

3.3 Verification

Traditionally testing has been carried out to verify that a software system is working as specified and there are not too many known defects. The problem is that while

testing can prove that there are defects in the software, it can not prove that there are no defects (Graham, van Veenendaal, Evans and Black, 2006, 18). It is impossible, because any sizeable software system will have so many execution paths and states that testing each and every combination is impossible. Therefore, testing is often focused on the most likely cases that can be derived from the customer requirements and specifications.

However, it is possible to take the requirements and the specifications, create corresponding test cases and execute those tests either manually or automatically in order to show how well the system fulfils the requirements and matches the specifications. Different types of tests are used to test different levels, as will be shown in chapter 4.2 and especially in Figure 2. Verification is often a very important part of testing, since the customer acceptance and ultimately the revenue are dependent upon it.

3.4 Quality Control

By executing tests and collecting and analysing results, it is possible to estimate the overall quality of the system. This information can be then used to guide decision on moving on to the next phase of the project, staying in the current phase or returning to the previous one. It can also act as an input to process improvement initiatives. Test execution is only a part of the quality control as it can include inspection of test plans, design documents and source code among other things (O'Regan, 2002, 23).

For quality control to work, the test suite needs to be well defined and repeatable. Controlling quality depends on the ability to draw trends on test result data, therefore low signal-to-noise ratio on tests and wildly variable test cases make the analysis hard. A reliable analysis also needs data collected over a relatively long period of time. With too short timeframe, there is not enough time for a noticeable change in the results to occur and small anomalies in the result data might be misinterpreted.

3.5 Regression Testing

The goal for regression testing is to ensure that already fixed defects do not get reintroduced to the system and that the other defects have not been introduced

(Grubb and Takang, 2003, 212). Especially when multiple versions of the system exist and are maintained at the same time, risk for the regression is high. This is because code is being developed in multiple version branches of the software and then merged to others. The situation is usually under control as long as none of the customers has to switch to a different branch. When this happens, for example in order to upgrade to a newer version, not all fixes for defects might be present on the new branch.

In this model, tests are created to detect issues raised by customers, business owners or other developers. Since testing is reactive, it is automatically focused on the areas where most of the problems are. This type of testing might not detect issues before the code containing them is deployed into customer's system and therefore needs support from other types of testing. Also, if the amount of defects is very high, the customer will have a negative view on the quality of the software.

Regression testing shines when the system is maintained and supported for a long time and new versions are regularly deployed in the production. A regression test suite can be built by writing tests that cover each and every discovered defect and executing these tests after the code is modified (Grubb and Takang, 2003, 213). Test suite will grow over time to cover the parts that customers find the most problematic. Regression tests automatically focus on that area and ensure that features that are most vital to customer are working correctly.

3.6 Measuring Maturity of the System

Tests and test plans can be used to measure the maturity of the system. The requirements can be linked to test cases and the results of those test cases give feedback how well the requirements have been fulfilled. In the beginning of a project most test cases either fail or can not be executed. As the project progresses and the maturity of the software grows more test cases can be successfully executed.

However, the tests linked to requirements do not give the whole truth regarding to maturity of the software. The amount of defects is a part of the maturity level too. Regression tests give good information that can be combined with the results of the tests derived from the requirements in order to measure the maturity of the software.

3.7 Summary

There are many reasons to perform software testing, ranging from simply verifying that the software is fulfilling all the requirements placed on it to trying to prevent costly errors. All testing performed should have a clear defined target that is measurable. Recording results of the tests and keeping them for the future allows graphing various variables and measuring how the maturity of the software changes as a function of time. One major reason for testing is that we are not able to formally prove that the software works (Grubb and Takang, 2003, 206).

Different types of testing might be performed at different stages of the software lifecycle and selected types depend on the software in question. In a simple desktop game there probably is less chance for costly errors compared to a software used in a space probe.

4 Automated Testing

4.1 Reasons for Automated Testing

Automating various things is always tempting. Automated systems can repeat tasks tirelessly, without mistakes and around the clock, leaving more interesting tasks for people. However, everything can not be automated and the cost of automation can be extremely high in some cases. Hass (2008, 362) identifies following cases where automation may help solve problems:

- Work that is to be repeated many times
- Work that is slower to do manually
- Work that is safer to do with a tool

Some specific types of testing are not possible to do with manual testing. For example performing a load test that simulates thousands of concurrent users would not be feasible to do manually. Based on the experience of the author of the present thesis, automating a test quite often takes more time than running the test manually. In addition to initial effort, the test case needs to be maintained when the

software system continues to grow and evolve. When a feature changes, test cases testing it might need to be updated and sometimes some even completely removed. Therefore, automating a test that is run only few times during the lifetime of the software system, might not be cost effective. However, in their book Fewster and Graham (1999, 3) have reported 80% decrease in costs of testing due to automation.

Most of the time test cases are faster to execute automatically than manually. However, the situation might change if the time required to write the test case and maintain it is taken into account. Tests that are executed only very few times during the application lifetime might not benefit from the automation effort in terms of saved time. Some tests require a large amount of data to be generated and are excellent candidates for automation. Generating that large amount of test data would be tedious, error prone and slow if done manually.

Computers are good at doing things exactly like told. This means that as soon as a test has been automated properly, it can be repeated over and over again. Computer will not make mistakes because of carelessness or being tired. Therefore executing complicated tests that require precise calculations are good candidates for automation.

In his bachelor's thesis (Koskela, 2012, 10) identifies collecting and reporting of test results as one of the advantages of automated testing. With sufficiently large amount of test cases, manually recording results and reporting them is both slow and error prone.

In their article Thomas and Hunt (2002, 38) voice suspicion that many developers have a feeling of instability and imminent danger every time they alter code. Having extensive automated tests in place helps to mitigate this feeling. This in turn lets developers focus on their specific tasks, without the need of keeping track of the whole system while they work on the code.

4.2 Cost of Change

Sooner or later in the software development project there will be a request for change. The most common reasons for these requests are defects, changed business domain, planned improvements and better understanding of the problem that the software tries to solve. Changes to the software need to be done in a structured

way, making sure that the system is still working as expected, otherwise the overall quality of the system will slowly degrade.

(Osborne, Brummond, Hart, Zarean and Conger, 2005, 20) suggests that:

> All design elements and acceptance tests must be traceable to one or more system requirements and every requirement must be addressed by at least one design element and acceptance test. Such rigour ensures nothing is done unnecessarily and everything that is necessary is accomplished.

Figure 2 shows the connection between different levels of definition and validation. It is worthwhile to notice how upper parts of the process are further away of each other than lower parts. This represents the difference in time: the time from laying out the initial user requirements to acceptance testing is longer than the time from detailed design to integration, test and verification.

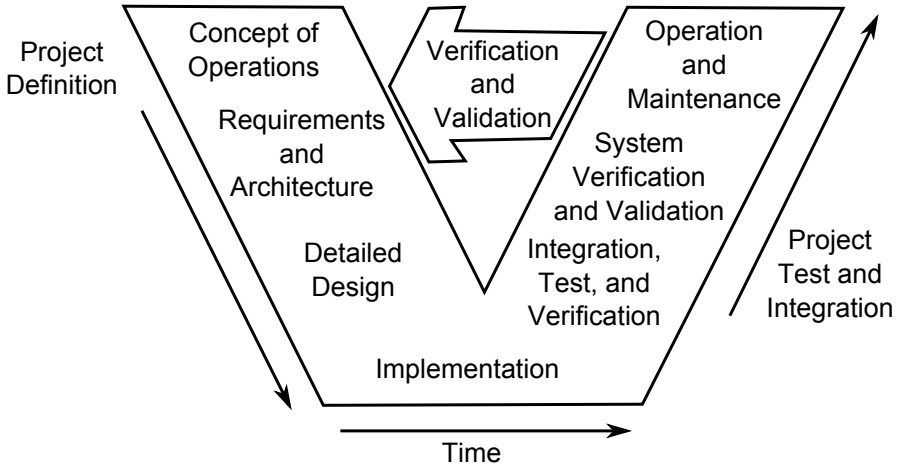

Figure 2: Systems Engineering Process (Osborne et al., 2005, 20)

The design done in the lower parts of the process depends on the design done at the upper parts. Essentially this means that changes done at the upper part will potentially affect the lower parts and require that appropriate testing is carried out.

Finding the Defects That Matter (Loveland et al., 2004, 27) identifies the amount of people involved in defect as a major factor of cost:

> A big piece of this expense is the number of people who get involved in the discovery and removal of the bug. If a developer finds it through his own private testing prior to delivering the code to others, he's the only one affected. If that same problem slips through to later in the development cycle, it might require a tester to uncover it, a debugger to diagnose it, the developer to provide a fix, a builder to integrate the repaired code into the development stream, and a tester again to validate the fix.

In their article Thomas and Hunt (2002, 36) underline a very good wisdom that combining many faulty components to a complex system is a recipe for disaster. It is advisable to start the testing effort as close to the source as possible.

If all the testing is done by people, manually executing test cases, time from specification to delivery will be long and the v-shape will be very wide. Automation seeks to bring ends of the v closer together by shortening the feedback loop. Instead of waiting for somebody to test that his change did not break anything important in the software, the developer can execute automated tests and get quick feedback about his change. If he finds out that some very rarely needed customer requirement that he did not remember to consider is broken, he can immediately start working on fixing the situation. Without automation, the developer gets his feedback when testing can be done manually and it can be rather hard to pinpoint the change that caused the test to fail.

In their book, Whittaker, Arbon and Carollo explain that over-investing in end-to-end automation tests often cements a system's design early on. The larger the automation suite is the harder it is to maintain. Time used to maintain brittle test cases could instead be used to improve the quality of the system (Whittaker, Arbon and Carollo 2012, 28.) This showcases the difficulty of test automation well; too little testing is not enough to help developers in their daily work and too much testing is hindering their work instead of helping it. Striking the balance between two ends is a hard and important task that needs to be paid constant attention to throughout the development of the system. As the time progresses, the needs of the testing change too.

4.3 Design

Automated testing is not always just about trying to find errors or verifying that the software meets the requirements of customer. Testing can also be used as a tool for learning more about the problem domain, components required by the software and their needed interactions. This type of prototyping and experimenting can be helpful especially in the beginning of the development, when a solid architecture has not yet emerged.

In order for the software to be easily testable, it generally needs to be loosely coupled, well designed and correctly divided into sub-systems, modules and classes. If appropriate rigour is shown during development and most, if not all, of the code is tested automatically, the design tends to be more flexible and easier to maintain than if the tests were written to only part of the code. This stems from the requirement to be able to instantiate the system under test easily in test harness with well defined inputs and outputs. Freeman and Pryce (2010, 229) talk about listening to tests in order to detect so called *"code smells"*, which are various common problems with software design. For example, if tests of a completely unrelated feature tend to break after change in software, there might be an undesired or unknown dependency in the software. Another common example is a class that is either hard or tedious to get in the test harness. This might indicate that the class is trying to do too many things and therefore has many dependencies.

If automated testing is not applied right from the beginning, the effort of testing the system gets harder and harder as the time passes. What might have been a simple test in the beginning of life-cycle of the system suddenly looks complex, ugly and hard to do. Whole books have been dedicated to presenting tools to solve this problem, one of the most notable being Feathers (2011). Automated testing is not a lost cause in such cases though, although it might require a slightly different approach.

4.4 Refactoring

Refactoring is the act of making small modifications on code in order to improve its quality, without changing the behaviour of the system (Fowler, Beck, Brant, Opdyke and Roberts, 1999, xvi). This is done in order to improve the internal structure of

the system and to facialite easier changes in the future. Ideally, the system is kept as close to working condition as possible during refactoring and tested extensively after each and every modification. This is nearly impossible with manual tests, so investment in automated testing is in order.

Refactoring is an integral part of test driven development (explained in more detail in chapter 11) and automated tests are essential for developer being able to change the code with confidence. By executing automated tests after each small change, developer has a greater confidence that his changes did not break anything unexpected.

The changes done in refactoring should not affect to any public interfaces, i.e. only the internals of refactored code is changed. The changes are also very small. A developer might change the name of a variable and run tests to verify that everything still works. Then he could extract a piece of code from inside a function and make another function to replace that functionality. And again he would run tests to see that everything still works. By taking small baby steps, the developer will know immediately if any of the changes breaks the code and fixing the problem will be easier than if the changes had been large.

4.5 Summary

Automated testing partly extends manual testing. Some of the things that are possible with manual testing can also be performed automatically. While the automation is faster and the test cases can be executed time after time, it is not free of cost: setting up an infrastructure to support testing takes time and money and the test cases need to be written and maintained. Testing can be performed faster and with smaller cost when automated testing is done correctly and focused to appropriate locations of the software.

Automated testing can be used to help the design of the software. Generally a software that can easily be tested with automated system is loosely coupled and modular. A software system like this is easier to change and maintain than a system that is tightly coupled and monolithic. Having ability to execute automated tests really fast generally helps the developers to maintain and refactor their codebase. This gives the developers confidence to do even bigger changes to code, without a nagging fear that they missed something crucial when making the changes.

5 Types of Tests

5.1 Motivation

There are many different types of tests and the distinction between them tend to be a little bit blurred. This is complicated by the fact that people tend to give names to things and hang on those names. A very specific classification between different types of tests is useful when experts of testing are communicating with each other, but for less specialised people the distinction does not need to be so important.

This chapter introduces 3 types of tests and defines their meaning in the context of the present thesis. This is done because what one developer might consider integration test, another developer would regard as an end to end test.

5.2 Unit Tests

Unit tests are usually considered being used to test the smallest scope of the system. They focus on only few objects or functions at a time and aim to test them. Because the system under test is usually really small, tests are fast to execute and hundreds of tests can often be run in few seconds. The small scope places some limitations on what unit tests can do and what they can not. Access to shared resources like file system, databases and network interfaces is often avoided and special components is often created to get around the limitations. These components include, but are not limited to, stubs, mocks and fakes and are treated more closely in chapter 9.5.

Because unit tests are testing the smallest pieces of the system, they tend to look at matters from a very technical point of view. It is not uncommon to write a test to verify that a function will return certain value when called with certain parameters. If such a test fails, pinpointing the source of the error can be really fast and the fix for it tends to be very local. Listing 1 is an example of a unit test written in Python. It creates two objects: model and listener and then registers the listener with the model. As a final step, the model is verified to have the event listener correctly set up.

```
def test_registering_event_listener():
    model = Model()
    listener = mock()

    model.register_event_listener(listener)

    assert_that(model, has_event_listener(listener))
```

Listing 1: Testing registering event listener

These tests are valuable for developers when they are working on the software; however, they give very little information to business owners and product managers. Their main purpose is to help developers with the internal quality of the software system. They test methods and functions directly and give a great deal of indirect information regarding the state of the source code: e.g. are classes easy to use in isolation, are functions short and to the point, are there not too many dependencies. Freeman and Pryce (2010, 229) call these clues *"test smells"* and instruct developers to actively pay attention to them. By listening to the tests, developers can improve the quality of the code and make the maintenance easier in the future.

5.3 Integration Tests

Integration tests have a broader scope than unit tests. They exercise a much broader part of the system and often make calls to database or access services on other computers. These tests are much slower than unit tests, however cover a larger part of the system. If integration test fails, pinpointing the source of the error can be more time consuming than in the case of unit tests because of the amount of code that is being exercised.

Depending on the context, the results of these tests can be understood by business owners. A test could for example verify that interest can be calculated correctly for a given customer and account. Listing 2 is an example of an integration test written in VB.Net. The test first creates a customer object and then saves it to the database. There is no explicit verification part; however, the test is deemed successful if saving does not cause an error.

```
<Test()> _
Public Sub TestSavingCustomer()
    Dim customer = CustomerBuilder.Create() _
            .withName("Test Customer") _
            .withNationality("Finnish") _
            .build()
    customer.Save()
End Sub
```

Listing 2: Testing saving a customer

5.4 End to End Tests

End to end tests are the largest of the three types of tests. They may exercise even a larger part of the system than the integration tests and their focus is already on the business level. These tests are the slowest to execute and they offer good a medium for business owners and developers to communicate with each other. These tests can often be derived directly from the customer requirements and can be written with a tool that supports processing natural language.

Koudelia (2012, 54) presents an example shown in Listing 3 for a behaviour driven test, which can be used to express behaviour of the system on a very high level. It describes four different passwords that are given to the system for verification and their expected outcome. This description itself does not specify what methods are called or how the results of the verification are displayed. These details are hidden out of the sight, because they would just add unnecessary complexity to the test.

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

Listing 3: Testing password validation

End to end tests are important, as they are used to verify what matters in the end: functionality of the whole software system as it is presented to the end user. These

tests ultimately verify that the system works as the end user expects it to work. If these tests are faulty, either not testing correct things or testing them incorrectly, the software system might not be what the customer wants it to be. Because customers are usually paying for the software, getting these tests right or wrong can have a direct effect on the future of the people writing them.

5.5 Summary

There are many kinds of tests, testing a system from different points of view and giving different kinds of reports about the state of the system. They all have their own strengths and weaknesses. A single type of tests usually is not enough to verify the correctness of the system. They are complementary in a sense that while looking at the same problem from different angles, they verify different aspects of the system and together produce a comprehensive estimate about the current correctness of the system.

It is important to identify why testing is being carried out and choose appropriate tools for it, before investing a great amount of money and time into them. If developers are already producing really high quality code and the biggest obstacles are in getting developers to understand business rules, high level acceptance tests might be a good solution. On the other hand, if developers already understand business well, but are having hard time in integrating their components together, integration tests might be helpful.

6 Amount of Testing

6.1 Motivation

The amount of testing required is a controversial subject. In traditional development models, especially in waterfall, testing is one of the last steps and usually lasts only as long as there is budget left. As soon as the budget has been spent, testing is stopped, regardless of the results or state of the system. The question about the amount of testing is actually threefold: what, how much and how often.

6.2 Focusing Testing

The most important question when testing is the question what to test. If there is enough time and money, everything in the software could be tested; however, usually this is not the case. Therefore it is important to focus the testing on finding the defects that matter.

The first candidates for automated testing are issues found by testers and customers. Generally, the defects raised by customers are the defects that matter most, otherwise they had not bothered to mention it. By creating an automated test before fixing the issue developers can ensure that an identical issue is never raised again.

Other good candidates for automated testing are new features. These have been deemed useful enough to be implemented and somebody is most likely paying money to get to use them. Here the goal is to verify that the features work as intended and catch possible issues before they are shipped to customers.

Knowing how the system is structured can help when choosing where to target the automation effort. Modules that are known to be central or very complex are good candidates for testing. Another good focal point can be found if there are few core modules that contain often used business logic and defects in this code would affect large portion of the functionality of the software.

Even when talking about automated testing created by developer, it pays to keep in mind the following: executing tests automatically may be fast and cheap compared to running the same tests manually; however, writing and maintaining those tests cost time and money.

6.3 Deciding on Amount of Tests

When testing is done automatically as a part of development process, the situation is somewhat different. Instead of spending what ever is left of a budget in the end of the project for testing, testing as part of development needs to be taken into account from the beginning. Automated testing and development are very interleaved, especially when dealing with unit tests, and it does not make sense to write detailed testing plans for this type of tests. Writing a good and detailed plan

for testing something that only exists as an idea, even if that, is impossible. Instead of that, there can be rules like *"One test case for each public function"* or *"One test case for each best-case scenario"* and *"One test case for each bug discovered"*. Rules like these can be useful, if they are decided based on facts and are mutually agreed and followed.

Modern tools are capable of analysing execution of tests and produce various reports that state how large percentage of statements of code is covered during tests and even show what sections of the code are executed. It might be tempting to say *"We need to have code coverage of 85% before we will ship the product."* This can be detrimental for the quality of both the code and the tests, because decisions like that guide testing and development to the wrong direction. Marick (1999, 8) explains how people tend to optimise their performance according to how they are measured, because often those measurements are used to decide how incentives are handed out. With criteria like this, there might be 85% code coverage; however, the quality of the tests is not necessarily very good. It is also worth remembering that statement or line coverage is a very narrow criterion. Kaner (1996, 7-13) lists 101 different types of testing coverage that will detect different kinds of errors. It would be foolish to focus only on one of them and leave others outside of any consideration.

Analysing statement coverage is better, if the number can be broken down to sub-systems, modules, classes and methods. This way the numbers can be analysed and cross-referenced with bug-reports, resulting with a rough idea where it would be good to have a look. If there is a sub-system that had much more reported bugs than any other sub-systems and there exists a central class or two that have very few tests, it might be a good idea to analyse that class more and see if it makes sense to do something about it.

6.4 Execution Interval

Tests run by an automated system are usually constrained by time and availability of hardware. As the amount of time needed to run the test suite grows, the amount of times they can be executed during a working day goes down. By dividing tests into different suites according to their focus and execution speed, a team can create a staggered solution for testing. Fast tests are run more often than slower ones and

offer the quickest feedback. Slower tests are run less often and their results complement those of the faster tests.

If a team is doing test driven development (see chapter 11 for more details), tests or a subset of them is run often, every couple of minutes as a part of development process. These tests need to be fast, because if they take approximately more than 30 seconds to run, developers will stop running them after each code change (Meszaros, 2007, 15).

A modern source control system offers a possibility to use hooks to perform actions before or after a change has been commited. It is possible to run unit tests automatically before each and every commit and abort the operation if they do not pass. This can be used as an additional safeguard against accidentally introducing bugs into code in source control. Again, this needs careful balancing, since even short delays in the very core part of developer's work are undesirable. If the team is already doing rigorous test driven development, this step might be superfluous and would only slow down development.

If the team has access to a continuous integration system (see chapter 12 for more details), tests or subset of them are executed after a suitable amount of changes have been commited into version control. Some continuous integration systems can be configured to run a build when there are untested changes and there have not been new changes during a given time. This can be used to group several changes into a single build. This is often the first build and test cycle that collects all the changesets together for a single build and so it is the first step to verify the work of the team as a whole.

It is possible to schedule tests to be executed at a given time. If the test suite is slow, it might be run during night against all the changes done during previous day. Essentially the team would be getting feedback on what they did one day later. Depending on the case, this might be sufficiently soon, especially if compared with manual testing where the feedback loop can be even longer.

The final possibility is to start test execution manually when the moment has been deemed to be right. This has the advantage that the person triggering the process can use his judgement and ask other developers if they are about to finish

something that could be tested. A drawback is that if nobody has time or remembers to start tests, they are not executed.

Thus, the type of the execution affects the interval how often the tests are executed. Various test suites take a different amount of time to execute and based on that, they can be selected to a specific type of test execution. A suite that can be executed really fast is a prime candidate for being executed as a part of test driven development, whereas a long running suite is best run during the night.

6.5 Summary

Deciding how many test cases to write, where to target them and how often to execute them is crucial for the testing effort to succeed. A large amount of test cases is often more costly to write and maintain than smaller amount and does not necessarily perform any better. Focusing the testing to the most crucial parts of the software will yield better results than testing without well thought plan.

There is always a compromise between cost and amount of test cases. Similarly, the decision of how often test cases are executed needs to consider costs and benefits. If the tests are executed rarely, the developers do not enjoy immediate feedback regarding to their changes. On the other hand, executing tests require resources like processor time, databases and perhaps dedicated hardware that all cost money. The developers can identify what tests are the most important and offer the most important feedback to them and execute those tests more often. Rest of the tests can be executed less often.

7 Anatomy of An Automated Test

7.1 Motivation

This section will present a general outline of a good automated test and the reasoning behind it. It will also go into details how tests were implemented in the case study to achieve this. The section focuses mainly on unit and integration tests.

7.2 Arrange, Act, Assert

A good test has three distinct parts: arrange, act and assert, commonly referred as 3A. In arrange part the system under test (SUT) is set up to a known state, in act the system is exercised and finally the assert verifies that everything worked as expected. There are multiple ways of doing each of the steps, none being always superior or the only right solution. The persons writing the test need to use their judgement and prior experience to choose a suitable method.

7.3 Focused Arrange

The ideal arrange part is short, focused on the relevant objects and showing only a necessary level of detail. When the system under test is simple and the objects are not composites of multiple other types, this can be easy to achieve. When the objects are very complex and have multiple values that need to be set, a simple arrange is not enough anymore. The examples in Listings 4 and 5 set up the same type of object; however, they have a different way of doing it.

```
<Setup()> _
Public Sub Setup()
    Dim exhaustMeter = new ExhaustMeter(500)
    Dim tyreInspector = new DomesticTyreInspector()
    Dim rustDetector = new RustDetector(InspectionLevel.Regular)
    Me.vehicleInspection = new VehicleInspection(exhaustMeter, _
                                                 tyreInspector, _
                                                 rustDetector)
End Sub
```

Listing 4: Setting up vehicle inspection

```
<Setup()> _
Public Sub Setup()
    Me.vehicleInspection = VehicleInspectionBuilder.Create() _
                    .withExhaustLimit(500) _
                    .build()
End Sub
```

Listing 5: Setting up vehicle inspection, take two

Both accomplish the same thing, setting up a VehicleInspection object for a car with domestic tyres, exhaust limit of 500 and regular level of rust checkup. The difference between these two setup routines is that the former exposed all the gritty little details about the internals of VehicleInspection, while the latter shows only interesting parts (exhaust level, in the case of this test). All the other parts of the setup are hidden away inside of the VehicleInspectionBuilder. This is in accordance of don't repeat yourself - principle (DRY). When the creation of VehicleInspection object eventually changes, there is a chance that only the builder needs to be changed and tests do not have to be modified at all. Meszaros (2007, 411) calls using methods to create SUT as *delegated setup* and recommends them to prevent code duplication.

The second advantage in the latter example is that the setup is more precise and only presents values that are interesting. TyreInspector and RustDetector are both created with default settings and the focus of the test is most likely centred around the exhaust limit on ExhaustMeter - object. The test could be checking that an old and polluting car will not pass inspection. For such a test, inspection of tyres is fairly irrelevant.

Nothing prevents chaining builders and creating a complex object in the way shown earlier. The example in Listing 6 creates a Character object with fully setup ActionFactory, which structure is shown in Figure 3.

Again, the setup shows that important parts of the test are:

- Character
- ActionFactory and especially the MoveFactory
- Location of the character

The test is probably about the character moving around and it is used to verify that the location of the character changes correctly when a move is executed. The object diagram of Character is much more complex than what the test setup implies, but it is all details that do not matter from the point of view of the test.

```
def setup(self):
    self.character = (CharacterBuilder()
                      .with_action_factory(ActionFactoryBuilder()
                                           .with_move_factory()
                                           )
                      .with_location((10, 10))
                      .build())
```

Listing 6: Setting up more complex object

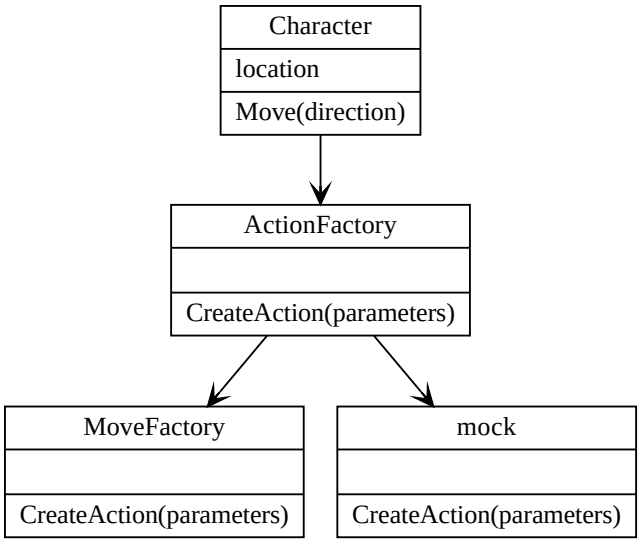

Figure 3: Fully setup Character with ActionFactory

7.4 Clear Assert

The assert part is where the state or interactions of a system under test are verified. As it is important to have a focused arrange part, it is equally important to have a clear assert part. A good assert is short, to the point and unambiguous. Again, it is often a good idea to hide the actual implementation details and write helper functions or classes to do the verification. If these helpers have interfaces defined to spell out what is being verified, the test is also easier to read. Chapter 8 approaches this subject from the point of view of domain-specific languages.

A very often used tool for writing clear asserts is Hamcrest. Hamcrest is a library for designing matcher objects that can be used for validation, filtering and testing (Denley, 2012). Hamcrest can be used to move the focus from little technical details, like attributes of objects to more domain focused testing. Listing 7 shows an

example, where pyHamcrest is used to verify that Pete is no longer hungry after eating some soup.

```
def test_eating_prevents_hunger(self):
        Pete = strong(Adventurer())
        meal = healthy(soup())

        make(Pete, eat(meal))

        assert_that(Pete, is_not(hungry()))
```

Listing 7: Using pyHamcrest for assert

The ability to give a detailed report why something did not match is a powerful feature of Hamcrest. The report contains information about what was expected and what was actually encountered. In case of very complex businness logic, this can help the developer to understand the problem better. Listing 8 has an example of a failed assertion that could result from the test in Listing 7.

```
AssertionError:
Expected: Character, who is not hungry (hunger factor less than 5)
     but: Character, who is very hungry (hunger factor of 45)
```

Listing 8: Failed Hamcrest assertion

Writing clear and understandable assertions does not depend on tools like Hamcrest though. With sensible structuring of the code, it is possible to write clear asserts by using the tools provided by the language and unit testing framework. One example how to do this is outlined in Figure 4. Meszaros explains that by extracting and encapsulating complex assertion login into a single function with an intent revealing name, the test suite is much easier to write and maintain (Meszaros, 2007, 475).

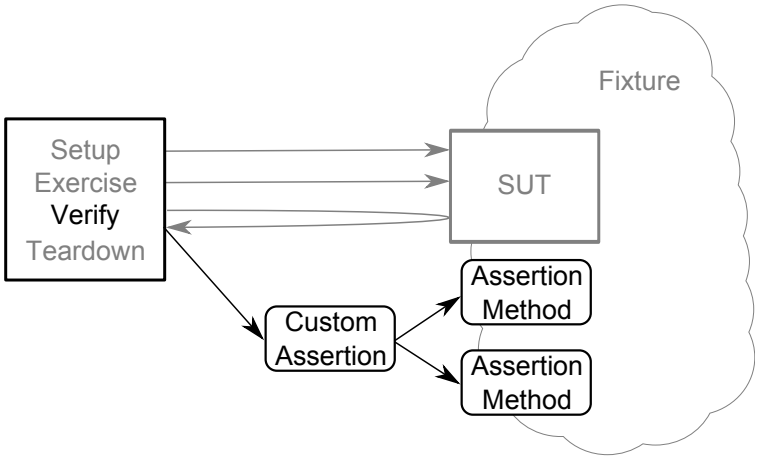

Figure 4: Custom assertion (Meszaros, 2007, 474)

One of the advantages of encapsulating complex assertion logic into a single function that has no side-effects besides failing a test suite is the possibility to test the logic (Meszaros, 2007, 475). This enables the developers to create common building blocks for tests that have been tested and verified to work. Naming the custom assertion using terms of the problem domain is a step towards a domain specific language, which are explained in more detail in chapter 8.

7.5 Summary

By following some guidelines and structuring tests to have distinctive parts for arrange, act and assert, the developers can create tests that are easy to understand and maintain. The test code should be treated with the same care and attention as the production code in order for it to stay maintainable.

While there are many tools that can be used to make the tests look nice and clean, there is no strict requirement to use them. Similar effects can be achieved by careful design and maintenance of the test code.

Code duplication can be reduced by extracting common logic appearing in multiple tests into helper classes and functions. These helper constructs can then be tested and verified to work correctly before taking them into use in tests. As the developers work on the infrastructure of the testing framework, they slowly create a common language than can be used in discussions regarding to tests and problem domain.

8 Domain-Specific Languages

8.1 Introduction to Domain-Specific Languages

Domain-specific languages (DSL) are languages that have been written for a very specific task. Common examples are Latex for document markup, Mathematica for symbolic mathematics and GraphViz for graph layout. In their book, Fowler and Parsons (2011, 27) define domain-specific language as a programming language of limited expressiveness focused on a particular domain.

Taha (2008, 1) explains how by trading functionality to expressiveness it is possible to create a language that is more accessible to general public than traditional languages. Often with very complex software systems the people writing the system do not really comprehend how it is supposed to be used and what the data handled in the system actually means. On the other hand, the people who understand the business domain very well often are not capable of translating that knowledge into code.

```
def test_that_hitting_reduces_hit_points(self):
        """
        Getting hit should reduce hit points
        """
        Pete = strong(Adventurer())
        Uglak = weak(Goblin())

        place(Uglak, middle_of(Level()))
        place(Pete, right_of(Uglak))

        make(Uglak, hit(Pete))

        assert_that(Pete, has_less_hit_points())
```

Listing 9: Testing behaviour with internal domain-specific language

Listing 9 shows a simple test case for an adventure game that has been written with an internal DSL. Many of the details have been hidden behind the functions that implement the test case, making it easier to understand the main point of the test. Only the assertion method is used from an external library called pyHamcrest, while everything else has been defined specifically for this test.

If this test case fails, assertion error is raised with informative explanation as shown in Listing 10. The error message contains information of which test failed, where it failed, why it failed and what the status of the object that caused failure was. It can be used by both domain experts and developers to effectively communicate and identify where to start looking for a possible error. Messages like this bridge the gap that often exists between domain experts and developers, because it gives them a common language that they can use to communicate.

```
FAIL: Getting hit should reduce hit points
----------------------------------------------------------------------
Traceback (most recent call last):
  File "nose\case.py", line 197, in runTest
    self.test(*self.arg)
  File "test_combat.py", line 51, in test_hitting_reduces_hit_points
    assert_that(Pete, has_less_hit_points())
AssertionError:
Expected: Character with less than 10 hitpoints
     but: Character has 10 hit points
```

Listing 10: Failed assertion

8.2 Types of Domain-Specific Languages

DSLs can be divided into two main groups: internal and external. The major difference is that while internal languages are essentially just a programming API that forms the language, the external languages have their own parser and syntax.

8.2.1 Internal Domain-Specific Languages

The earlier shown Listing 9 is an example of an internal DSL written with Python. It is readily apparent that the test has been written with a programming language, because of the usage of parentheses and other programming language constructs. However, the test hides many unimportant details with clever use of functions and objects and only shows those concepts that are important for understanding the test.

Essentially, an internal domain-specific language exists completely inside of the host language: the language the program is written with. Using constructs like method chaining as explained by Fowler and Parsons (2011, 373) it is possible to write code that looks closer to natural language than regular programs.

At the core of DSL is often a semantic model, which is a representation of what a DSL describes (Fowler and Parsons, 2011, 373). While not always necessary it is often a good idea to build a semantic model because having one makes it easier to move from internal DSL to external DSL. Using semantic model also makes it possible to test the semantics and populating the semantic model separately (Fowler and Parsons, 2011, 162). In case of very complex DSLs this advantage can make working with the model a much easier task.

Because internal domain-specific languages are embedded in the host language, they are most often also edited with the tools that are used to edit a program written in the host language. Working with internal domain-specific language might be confusing at first if there is no prior programming experience. This makes the developers best candidates to work with internal DSL, because the domain specialists might not have the required skills and experience.

8.2.2 External Domain-Specific Languages

External languages have their own syntax and parser that can be independent from the language constructs of the calling system. This gives greater freedom in designing the language; however, it also means more work because the need of parser. Listing 11 shows how to use Gherkin, which is a business readable, domain-specific language, to write a test case.

The test reads almost like a small story written in English, albeit with somewhat clumsy sentences. Concepts of the problem domain (securities trading) are in central role. One does not really need to know anything about programming in order to understand what is being tested. This is what makes external languages very powerful in bridging the communication gap between software developers, domain specialists and customers.

Because an external domain-specific language does not have access to capabilities of the host language they tend to be somewhat more limited than internal domain-specific languages. Each and every feature needs to be separately coded with support from both the parser and model tree. On the other hand, because an external domain-specific language is not bound by the limitations and design constraints of the host language the possibilities of the language are virtually

```
Feature: Orders
    In order to trade securities
    As a customer
    I want to be able to manage orders

Background:
    Given User has logged in
      And security "Test" is share
      And security "Test" has price 10

Scenario: Create a purchase order
    Given "Pete" is a person customer
      And "Pete" has a portfolio with an account
     When "Pete" makes a purchase order of amount 100 for "Test"
     Then "Pete" should have 1 open purchase order for "Test"
```

Listing 11: Testing purchase order with external domain-specific language

limitless. For example, it is possible to write a language that resembles natural language like Finnish or English and use it to specify tests.

8.3 Summary

One of the goals of both internal and external domain-specific language is to create a language that has a limited set of features and is easier to use inside of the problem domain. This language is easier to use than a general purpose programming language and allows non-technical people to work with the software. Because a domain-specific language can be understood by both the developers and the domain experts, it can be used as an ubiquitous language allowing more effective and error free communication. Domain experts can use the terms that they are familiar with in the context of the problem domain and the developers will have a handy dictionary that maps those terms directly into code. In most extreme cases, domain experts can use the domain-specific language to write test cases or configure how the software works in a specific situation.

Since internal and external domain-specific languages can share similar constructs like the model tree, it is possible to start with an internal domain-specific language and later on add a parser in order to support an external one. This splits writing an external language to two major tasks: creating the model tree and using it to drive

the language. Testing an internal domain-specific language is a little bit easier, since the intricacies of writing and testing a parser do not get on the way. After the structure of the semantic model and behaviour of the language is better know, the parser can be developed in order to support an external language.

9 Managing Dependencies

9.1 Motivation

A large software system may consist of hundreds or thousands of components that relate to each other in a way or other. Some components rely on others to offer their services. In such a case, it is said that a component depends on another component.

Managing dependencies of a large software system is a crucial task. Without proper attention to it, the codebase will slowly deteriorate and dependencies between components will get out of hand, making expanding and maintaining the system a nightmarish task.

9.2 Inversion of Control

Inversion of control (IoC) originally meant a programming style where an overall framework or runtime controlled the program flow (Seemann, 2012, 42). Programmer essentially relinquishes some control over his software to a framework or runtime. The framework might for example be used to control lifetime of objects in the system, their instantiation or calling specialized methods. The programmer does not have full control of the system anymore; however, he has gotten an easier environment to program with.

9.3 Dependency Injection

Dependency injection is a specific case of IoC, where dependencies are controlled by a framework. It is one of the many tools for writing loosely coupled code. The basic idea behind it is to construct a software system from loosely coupled components that are wired together at the startup of the system. Objects that create the software system can be thought to form an object graph, where components are connected to their dependencies.

Dependencies are best specified as interfaces and not as concrete classes. This enforces loose coupling between components and enables substituting one component with another, as long as they both conform to the same interface. Technical implementation may be an interface or abstract class, but the main principle is the same. This principle is named to Liskov Substitution Principle (LSP) after Barbara Liskov, who presented it in her presentation at Conference on Object-Oriented Programming Systems, Languages, and Applications in year 1987. The original definition as presented by Liskov (1987, 25) is as follows:

> If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T.

LSP is one of the core principles in modern software development and located in the heart of dependency injection.

Dependency injection is a very pervasive design pattern and therefore it is very hard to refactor an existing application towards it as pointed out by Seemann (2012, 42). Dependency injection starts at the very top of the application and reaches the very bottom depths of object hierarchy. It is not an impossible task; however, it requires good support from comprehensive automated test suite and a great deal of hard work.

Like with all patterns, there are anti-patterns relating to dependency injection. Seemann (2012, 135) identifies four most common ones as: *control freak*, *bastard injection*, *constrained construction* and *service locator*. Control freak pattern is almost an opposite of dependency injection, because instead of injecting dependencies from outside, they are created by the object using them and thus making them hard coded dependencies. The pattern is easy to distinguish, but hard to fix if the system is large and complex. More on how to tackle this kind of challenge can be found in chapter 10.1.5.

In the heart of the service locator pattern is a central registry where services are located. This registry is made available to all of the program usually by a global variable or singleton pattern. The caller can then use the registry to retrieve a fully instantiated and configured service. Because the registry is available everywhere

dependencies of any given module are not visible. Even worse is that the client code depends on the central repository and cannot function without it. This makes code reuse harder, because all the components of the system have to use the same kind of service locator.

Service locator is similar to a pattern called abstract factory. In abstract factory client code can request an object to be created with given parameters. The difference is that while there is no limitation on number of abstract factories the system can have at any given time, there usually is only one single service locator.

9.4 Dependency Injection Container

Dependency injection container (DI-container for short) is a framework or library that can be used to compose object graphs based on their dependencies and configuration. Instead of composing an object graph manually, the developer can simply request a DI-container to compose one for him.

DI-container may look similar to the abstract factory or service locator; however, it is quite different in the fundamental level: the code never requests for a dependency, rather it is forced to consume it (Seemann, 2012, 7). This distinction is very important, because it is easy to misuse the DI-container and end up with an implementation that is nothing more than a glorified service locator.

In general, DI-container should be used as close to the application entry point as possible (Seemann, 2012, 75). The rest of the application has access to the services it needs because they have been injected via constructors when the object graph was resolved by DI-container. Essentially this means that only a very small part of the system, called as *composition root*, should be aware that DI-container even exists.

As part of composing object graphs, DI-container can also take care of the lifetime management of components. This allows software to be configured to reuse component instances that are expensive to create or need to share information between threads.

9.5 Dependencies in Tests

Dependencies between components and external systems can make testing really tricky or even impossible. This is true especially if the dependencies are static i.e. not injected from outside. If dependencies are handled outside of the component that requires them to work properly, situation is much easier. A database is a common example of such a dependency. Connecting to databases, ensuring that the data is in correct state and cleaning up after testing can be a tricky and time consuming process.

A common solution is to use some kind of a fake or mock object that looks and behaves like dependency, in this case a connection to a database. A developer can specify how this object behaves and how it will respond to queries. This enables testing without using the actual database, making it both faster and easier. Figure 5 shows an overview of replacing a dependent on component (DOC) with a test double. Because SUT does not create DOC directly, but uses the component that has been supplied to it testing is easy and straightforward.

Test components pretending to be a specific dependency can be constructed in various ways and they have been called with multiple names: mock, stub, duplicate, fake and similar. In the present thesis the term *"substitute"* is used, because it is rather neutral and technical details between different implementations are not important.

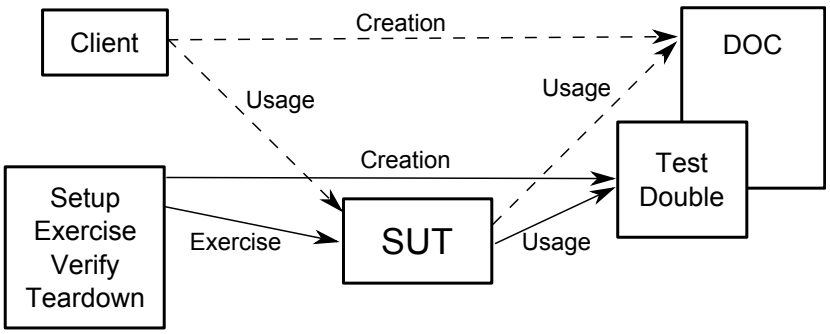

Figure 5: Test double injection (Meszaros, 2007, 70)

9.6 Summary

Inversion of control is a technique where a programmer gives up some of the control in exchange for easier programming environment. Dependency injection is a form of

IoC where dependencies are handed to the components from outside instead of being created by the components themselves. IoC promotes loose coupling and code reuse.

While IoC does not depend on any specific tools, there are some frameworks that can make dependency injection easier. These DI-containers can automatically create instances of required components and inject correct dependencies into them automatically. In addition to that they often can control life time of registered components and offer advanced features like lazy initialization.

Testing a component that is written to take advantage of dependency injection is simpler than testing similar component that creates dependencies by itself. Services like databases and file systems can be replaced with substitutes during testing in order to have a full control of their behaviour. This also can improve the performance of the tests quite a bit.

10 Legacy Code

10.1 Challenges Presented by Legacy Code

The following chapter takes a look into some of the challenges that testing a legacy code presents. Instead of trying to cover each and every possible problem some interesting and varied cases were selected. The selection is partly based on the personal experiences of the author of the present thesis.

The common theme to all these challenges is that they make developing the new features and maintaining the old ones slow and error prone. Each and every change that is done needs to be meticulously tested and verified in order to avoid bugs. Still the developers often have that nagging feeling that they forgot something while making the change and sometimes seemingly unrelated part of the software stops working correctly.

10.1.1 Original Developer Left And Did Not Leave Documentation Behind

Sometimes software systems are developed over a span of decades. New features are added and existing ones are removed or changed. Tools and techniques used might change over time; however, some of the code is left untouched and still uses old

technology. New developers are hired and old ones leave or retire. Even if the developers have good intention to keep the documentation up to date with all the changes, eventually it will fall out of sync. At this point tacit information starts to be valuable, but when original developers leave, the situation gets bad. At this point, there might exist a very complex software system that nobody really understands. Developers have fragmented information and changes are very risky and slow because of this.

```
def test_poison_causes_damage(self):
        Pete = strong(Adventurer())

        affect(Pete, with_(weak_poison()))

        assert_that(Pete, has_less_hit_points())
```

Listing 12: Test as an example of business requirement

In this kind of situation automated tests can work as an executable specification. After all, they specify start conditions, actions taken and expected outcome. Meszaros (2007, 33) mentions how tests can be seen as examples of how the system is supposed to work. Listing 12 shows an example of how test can be used as an example to communicate how poison should affect characters. This example is from the business point of view and does not go into little technical details.

A more technically oriented example is shown in Listing 13. It exposes more of technical implementation, while still being as terse as possible. The test is also focused on verifying behaviour instead of state. It specifies how Model and Character objects interact, when an effect is added and removed from Character.

The advantage that tests have as a documentation over traditional documentation is that they can be verified easily. Every time the test suite is executed, the tests are either passing or failing. As long as the tests are kept valid and passing, they can also be treated as a valid documentation about how the system is supposed to work. This is much easier to remember to do than updating a design document or programmer's guide.

```
def test_effect_expiration_event_is_raised(self):
    model = mock(Model)
    add_event = mock(Event)
    remove_event = mock(Event)

    character = (CharacterBuilder()
                 .with_effect(EffectBuilder()
                              .with_duration(0)
                              .with_add_event(add_event)
                              .with_expiration_event(remove_event)
                 .with_model(model)
                 .build())

    character.remove_expired_effects()

    verify(model).raise_event(add_event)
    verify(model).raise_event(remove_event)
```

Listing 13: Test as an example of technical implementation

10.1.2 Database Connection Inside of Business Logic

Often there is need for a business logic to access the data stored in a database. The most straightforward way of doing this is to open a database connection when needed, query the data and then close the connection when it is not needed anymore. Since opening a database connection is a rather slow operation, connections to the database are often pooled. There is a central location in the software system where client code can request a database connection. The problem with this approach is that while getting a connection to a database is easy for client code, there is a hard dependency on the central location where the database connections are retrieved. Setting up a database connection for each test is a slow operation and it increases likehood of tests interacting with each other.

The diagram in Figure 6 shows how ItemHandler class is using Item class to connect to the database. Because the database connection is handled inside of the Item class, there is a hard dependency for database. If the developer would want to test the ItemHandler class, the database would need to be set up, connected and populated with the test data.

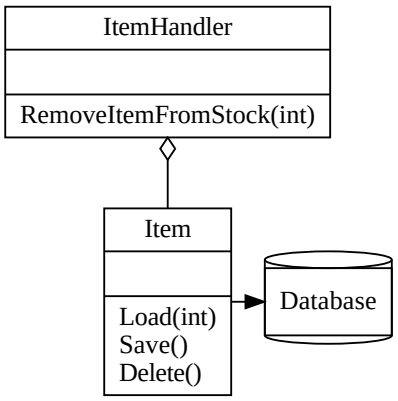

Figure 6: ItemHandler

Listing 14 shows an example code where ItemHandler class has a method RemoveItemFromStock. The method is used to load an item with itemID from the database, decrease the quantity in stock and save the item back in the database. Database connection that is used is retrieved from a static class called Application. Writing a unit test for RemoveItemFromStock method requires the developer to set up the Application class with session, database connection and the other things it might require. If the only method for configuring the Application class is via configuration file, the test needs to access file system too.

```
Public Sub RemoveItemFromStock(ByVal itemID As Integer)
    Dim item As Item
    Dim dbConn as Connection

    dbConn = Application.CurrentSession.Connection
    dbConn.BeginTransaction()

    item = New Item()
    item.Load(itemID)
    item.quantity = item.quantity - 1
    item.Save()

    dbConn.CommitTransaction()
End Sub
```

Listing 14: ItemHandler without repository

One possible solution to this problem is shown in Figure 7, which illustrates how ItemHandler class can be written using a simple repository pattern. The

corresponding code can be found in Listing 15. The basic idea is to remove basic database operations (creation, read, update and delete, commonly known as CRUD operations) from the business class and place them into a repository. The reposity now has the responsibility to perform all CRUD operations regarding to the Item class. Since the ItemRepository implements interface IItemRepository that exposes CRUD operations, it can be replaced by a substitute during testing. As a final step the constructor of ItemHandler class has a parameter for supplying IItemRepository that it is then used to access database.

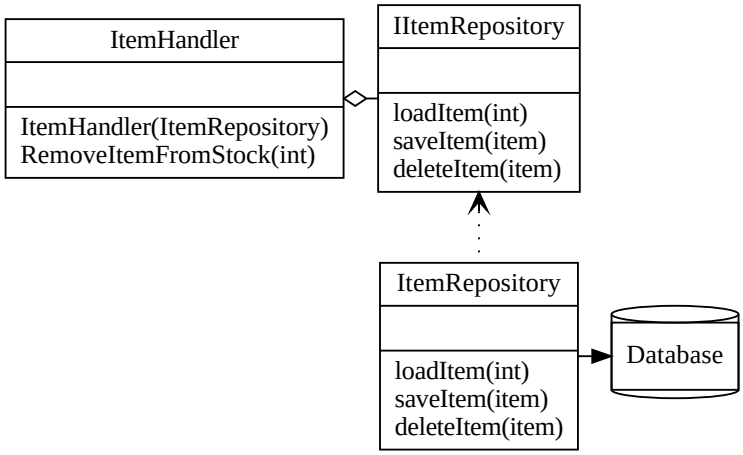

Figure 7: ItemHandler with repository

The corresponding code is shown in Listing 15. It is very similar than the one shown in Listing 14; however, some key differences are present. The method does not depend on a global variable or a static class. The database connection is abstracted behind IItemRepository and supplied to it from outside. These changes make it possible to test the method in isolation, without setting up the Application class or a database.

```
Public Sub RemoveItemFromStock(ByVal itemID As Integer)
    Me.repository.BeginTransaction()

    item = Me.repository.LoadItem(itemID)
    item.quantity = item.quantity - 1
    Me.repository.SaveItem(item)

    Me.repository.CommitTransaction()
End Sub
```

Listing 15: ItemHandler with repository

Nhibernate is an entity mapping framework that can be used to hide database details from business logic (Perkins, 2011, 2). On a surface the code in Listing 16 is very similar with the one using repository. The major difference is hidden out of the sight: in repository pattern the developer has to write the sql queries that update the database; however, the NHibernate can take care of all that after the developer has configured the mapping between database fields and object properties.

```
Public Sub RemoveItemFromStock(ByVal itemID As Integer)
    Dim session As ISession
    Dim transaction as ITransaction

    session = Me.sessionFactory.getSession()
    transaction = session.BeginTransaction()

    item = session.Get(Of Item)(itemID)
    item.quantity = item.quantity - 1
    session.Update(item)

    transaction.Commit()
    NHibernateHelper.CloseSession()
End Sub
```

Listing 16: ItemHandler with NHibernate

Adding NHibernate to a software system later than very beginning might prove to be a very tricky business. NHibernate works best when all the access to a database is done via it because sharing the sessions and database transactions with a legacy code is not easily possible. In the worst case scenario the source code of the application will be divided into two portions: old and new. The old side of the code can access database using legacy methods like direct sql-queries or business objects, while the new side uses NHibernate. This makes session management hard and sometimes a process has to be tailored to work around the limitations of the system and not the other way around.

Advantage of NHibernate is that it makes working with different databases really easy. The developers do not usually have to concern themselves with the differences of sql dialects and the code is more straightforward to write. The performance might suffer a little bit from using NHibernate; however, using various caching strategies can remedy that to a degree.

A third option that can sometimes be used is shown in Figure 8 and Listing 17. In this option ICommand is used to abstract the database connection and is supplied from outside during the call. This again makes it possible to substitute it during testing. Command can be used to execute commands and queries directly, without much abstraction like business classes. This method works very well in cases where database operations can be performed directly. The advantage of this approach is that the batch operations where a large amount of data is updated are relatively fast to perform compared to using business classes where objects are loaded, updated and then saved back into the database. A disadvantage of this method is that it scatters sql commands and queries all over the system. This makes changing the database schema harder, since the changes are spread over larger portion of the code.

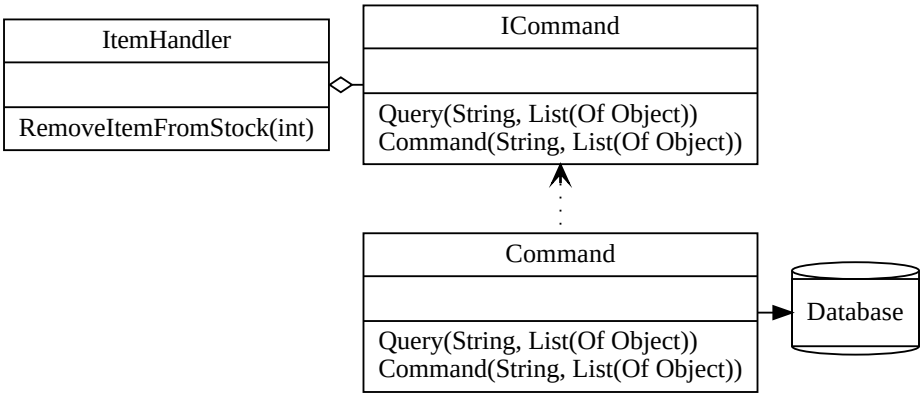

Figure 8: ItemHandler with Command

The code shown in the Listing 17 is again removing a single item from the stock with a given item identification number. This time the operation is performed directly in the database, without loading or saving Item business object. It is clearly visible how this approach scatters sql commands and queries in much wider area than using business objects. However, depending on the design constraints of the system this may be acceptable.

10.1.3 Static Methods Guiding Execution of Business Logic

Sometimes a configuration file is used to control how the software works in specific situations. It is tempting to place these values in a static class where they are easily accessible. The problem with this approach is that it introduces a hard dependency that is hard to substitute and is not visible from outside of the client method. If

```
Public Sub RemoveItemFromStock(ByVal itemID As Integer,
                               ByVal command As ICommand)
    command.BeginTransaction()

    parameters = New List(Of Object)({itemID, itemID})

    command.Command("UPDATE ITEM SET QUANTITY = " +
                    "(SELECT QUANTITY FROM ITEM WHERE ID = ?) - 1" +
                    "WHERE ID = ?", parameters)

    command.CommitTransaction()
End Sub
```

Listing 17: ItemHandler with command

multiple such methods or classes are used, the amount of tangled dependencies start to grow and the software soon turns into a maintenance nightmare.

```
Public Function Distance(ByVal start As Integer,
                         ByVal destination As Integer) As Integer
    Dim difference = destination - start

    If difference < 0 AndAlso Configuration.NoNegatives Then
        Return 0
    Else
        Return difference
    End If
End Function
```

Listing 18: Static control logic

Listing 18 shows a simple routine that is used to calculate the distance between two points on a line. The system can be configured to ignore negative values and return 0 instead. The problem here is that in order to test this simple function, the developer needs to set up static Configuration class with correct parameters. In a very bad case, the Configuration class can only be instantiated with values loaded from a database, so testing the routine also needs the database. Setting up all these dependencies just to test a simple routine is slow and cumbersome.

There are multiple possible solutions for this problem. A simple one is to add a new parameter that can be used to inform the function on whether it should return

negative values or not. Implementation of this is shown in Listing 19. In this simple example this solution works just fine and now the function can be tested in isolation.

```
Public Function Distance(ByVal start As Integer,
                         ByVal destination As Integer,
                         ByVal noNegatives As Boolean) As Integer
    Dim difference = destination - start

    If difference < 0 AndAlso noNegatives Then
        Return 0
    Else
        Return difference
    End If
End Function
```

Listing 19: Adding a control parameter

This approach works when the function does not have multiple configuration parameters. In the case of complex business logic the function's parameters list would quickly get big and unwieldy to use. The code would not look particularly elegant and using the function would also be dangerously error-prone. Adding a new control parameter would also mean that all the locations where the function is being called from would have to be updated to pass the new parameter.

In our simple case it would be possible to write two different versions of the method and give them intent revealing names. The first one would return negative values while the second one would substitute them with a zero. In case of a complex business logic this approach would not work, because amount of combinations of different control parameters could be too high to easily maintain. A better approach would be to remove the static dependency and pass it from the client code. This helps in testing the function in isolation, although the caller still needs to get an instance of the Configuration class from somewhere. Example of this approach is shown in Listing 20

The end result is almost identical to the original code example shown in the Listing 18. By using an interface instead of a concrete implementation the developer has made the function easier to test. The IConfiguration can be substituted in tests easily and setting it up at the beginning of a test is a fast operation.

```
Public Function Distance(ByVal start As Integer,
                         ByVal destination As Integer,
                         ByVal config as IConfiguration) As Integer
    Dim difference = destination - start

    If difference < 0 AndAlso config.NoNegatives Then
        Return 0
    Else
        Return difference
    End If
End Function
```

Listing 20: Passing configuration

It is worth observing that IConfiguration should contain only the minimal amount of data. While it might contain only a value or two in the beginning it is possible that as the time progresses more values are added. Eventually the IConfiguration will contain a large amount of fields that are not really related to each other anymore and using the interface will not be easy anymore. The interface segregation principle states that the client should not be forced to depend on the methods of interface that it does not need (Martin, 1996, 5). The reason behind this is that if multiple clients depend on the same interface, they are in essence coupled together. If the client depends on methods of an interface that it does not need, that coupling is unnecessary. The interface could also use a better name like IDistanceCalculationConfiguration instead of just IConfiguration.

10.1.4 Huge Method That Does Everything

Huge methods that have a lot of responsibilities cause common problems with legacy software. Huge methods might have started out as a small, well defined functions; however, as the time passed and more functionality was added to the system, the new features were added by modifying the existing functions. These large functions are hard to maintain, test and debug, because they can be hundreds if not thousands lines long and contain loops within loops. Just understanding how the function is supposed to work can be a daunting task.

Because these huge functions have multiple responsibilities they tend to have lots of dependencies too. More often than not those dependencies are hard and can not be

easily substituted during testing. In order to make the situation more manageable the function needs to be broken into sensible parts that can be independently tested in isolation. The original function would then call these other functions to perform the same operations that it used to take care of all by itself.

Feathers (2011, 14) suggest to first cover the main functionality of the huge function with integration tests that capture the business requirements of the function. After this a developer can start slowly breaking the function apart into smaller chunks while verifying constantly that the integration tests are still passing. The small chunks that the developer creates by extracting functionality from the original function should be written in a way that they can be tested in isolation.

10.1.5 Control Freak

The control freak anti-pattern was already mentioned in chapter 9.3. Essentially it is a pattern where dependencies are not controlled from outside of the components, but rather from inside. Components are responsible for creating the dependencies they need and thus there exists a hardcoded dependency between the components. This makes automated testing harder and the structure of the software feels sluggish and hard to change.

Seemann (2012, 143) lists three steps that can be taken in order to refactor a system into a more suitable state:

- Ensure you are programming to an interface.
- Move the creation of a particular dependency to a single location and ensure that it is represented as an interface.
- Move the single location of creation outside of the class by implementing DI pattern, such a constructor injection.

If for a reason or another the constructor injection is too hard to do, one can apply a pattern called parametrised constructor (Feathers, 2011, 379). In this pattern, a new constructor is created that can be used to supply the dependency outside. The original constructor is kept around and default implementation of the dependency is created there. This allows fast refactoring in order to make unit testing easier;

however, it does not require changes at each and every location where the component is created.

Depending on the dependency being injected via parametrised constructor, this stage should be treated only as a temporary step towards full constructor injection. The component still has a static dependency and reusing it might drag along components that are not needed in the new system.

It is also worth noting that the whole object graph does not have to be refactored in one go. Depending on the situation, refactoring can be started either from the top or bottom of the graph and dependency creation is slowly pushed upwards, until it reaches the composition root. For some time, dependencies might look really ugly and big, especially if the refactoring was started from the top; however, eventually the situation will get better and the code will transform into a more readable and maintenable form.

10.2 Testing Legacy Code

Testing legacy code can be a tricky business. The code might have evolved over a course of years or even centuries in extreme cases and during that time it has faced many changes. Developers have changed, requirements are different now than what they were in the beginning and development paradigms have risen and fallen. Some of the temporary modifications done in a hurry were never removed and only half-understood methods litter the codebase. All these events have left their mark on the code and it is not as easily maintainable as it was when it was young. Interestingly these are also the most common excuses cited when asked why people do not want to test their code. Granted, some of them might be very valid reasons, but usually people tend to exaggerate the negative aspects in order to justify skipping the testing.

While working with the legacy code it is important to remember that a complete rewrite usually is not an option. The customers depend on the current version of the software and require maintenance and maybe even new features. Therefore it is prudent to first ensure that the software does not break because of the changes by writing sufficient amount of integration tests. After the integration tests are in place, the code can be slowly restructured to allow writing unit tests.

10.3 Summary

Working with legacy code can initially feel difficult because making changes is slow and prone to errors. The temptation to just abandon the codebase and rewrite the program from scratch might be quite big too. However, with a methodical approach and constant attention to the technical excellence the obstacles can usually be solved.

It is often useful if the developers can identify the most common problems in their codebase and come up with an agreed solution on how to avoid them in the future. Because there often are multiple ways to solve a single problem, it is important that the team has an agreement on how they will approach the probem. This way the codebase will not deteriorate further and this gives the team chance to start improving it.

11 Test Driven Development

11.1 Overview of Test Driven Development

Test Driven Development uses various tests not only to verify the code, but also to guide its design. Freeman and Pryce (2010, 6) suggest that *"As we develop the system, we use TDD to give us feedback on the quality of both its* implementation *("Does it work?") and* design *("Is it well structured?")."*

TDD breaks writing software into three distinct parts: test, development and refactor.

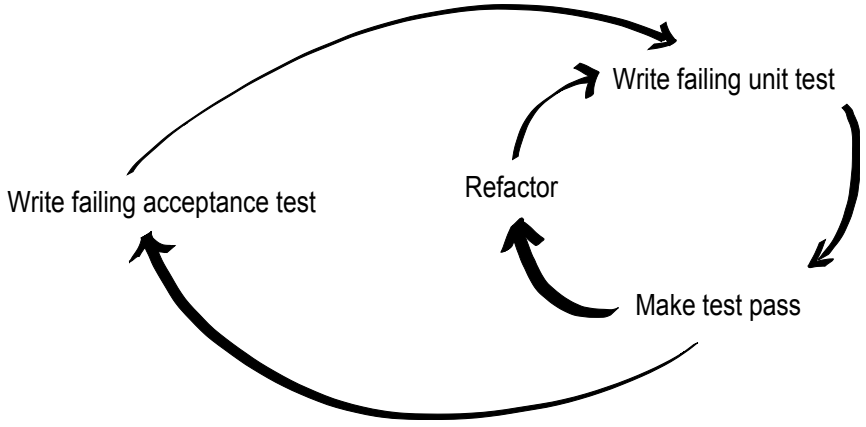

Figure 9: TDD in a nutshell (Freeman and Pryce, 2010, 6)

Adding a new feature starts with a new acceptance test being written. This test captures the user requirement for the feature. Developers will then start implementing the feature by adding a unit test identifying what they need to change first, making necessary changes and cleaning up the code as the last step. Then new unit test is added and the whole cycle repeats until the acceptance test shows that the user requirement is fulfilled.

If acceptance tests can be written in collaboration with customers (either end users or domain experts), they become a very important bridge between people with business domain knowledge and people with technical domain knowledge. In his master's thesis Koudelia recognised that for various reasons, customers are not very willing to invest enough time and effort in this (Koudelia, 2012, 81). This places a major burden on the shoulders of developers, since their task is now to read and understand specifications, translate appropriate parts of them to acceptance tests and write the software to pass those tests. Instead of this, a part of the specifications could have been written as acceptance tests, instead of a traditional specification document.

11.2 Advantages of Test Driven Development

Freeman and Pryce (2010, 57) list three aspects in test driven development that help in designing the software. By writing a test first, the developer has to consider what the object is supposed to do and this in turn helps him to manage the scope. Secondly, unit tests are supposed to stay small and compact. If they start to grow, it is an indication that the object in question is doing too many things and needs to be split up. Third, for an object to be unit testable, all dependencies have to be exposed and given to it from outside. This helps maintaining a loosely coupled system. Seemann has reached a similar conclusion and explains that TDD is the safest way to ensure that the system is testable (Seemann, 2012, 20).

Test driven design gives developers very fast feedback if the code they have written is working as it should be (Erdogmus, Morisio and Torchiano, 2005, 226). Ideally the first step to implement anything is to write a test that will show if the functionality is working so a developer always has a proof of how well the system is working. In practice this is not always possible, but the developers should try to minimize the amount of code that is not tested automatically.

Another advantage is that the developer is encouraged to decompose the problem into small, manageable tasks thus helping maintaining the focus and providing steady, measurable progress (Erdogmus et al., 2005, 226). Because the developers see steady progress being made they feel more encouraged to keep working even on very large tasks that take long time to finish. And having that large task split into smaller pieces helps them to focus on a single problem that they can easier manage and keep the details in their mind.

11.3 Challenges of Test Driven Development

Test driven development requires a significant investment of time during coding of a feature. While Erdogmus et al. (2005, 236) indicate increase in the productivity they also highlight that productivity seem to have increased variation. Erdogmus et al. continue and theorize that the variation might be due to relative difficulty of the technique and their research results indicate that too. If a product schedule is very tight the developers might feel an urge to cut corners and skip the automated testing part.

Another challenge is the maintenance of the tests. When the software system evolves and changes sometimes the tests have to change too. If the tests are written in a way that couples them very tightly with the production code, the need to change them arises more often than if they were loosely coupled. In a way the tests bring inflexibility to the design instead of making it more flexible. It is possible to use a domain specific language to alleviate this to a degree.

11.4 Summary

Test driven development interleaves testing and coding very closely and gives the developer constant feedback regarding the progress. As an automated test suite is built during the development of a feature it is easy to run those same tests to check for regression. An extensive test suite helps in refactoring because the developer can feel more safe when making changes in the code.

Test driven development requires strict discipline and understanding that the effort made on the tests will pay back both in short and long term. For example, cutting corners because of a tight schedule will often eventually backfire and the team ends

up spending more time compared to doing the testing and development correctly in the first place.

12 Continuous Integration

12.1 Introduction to Continuous Integration

Moreira (2010, 126) defines continuous integration (CI) as a process of integrating code frequently in order to reduce large integrations, complexity and to make functional software readily available. Cauldwell notes that the more often integration is performed, the easier it will be (Cauldwell, 2008, 22). This is because the amount of changes to be integrated is smaller and easier to manage. In case of problems in the software system after integration, the amount of changed code is smaller and it is easier to try and find the cause of error.

Holcombe (2008, 31) identifies CI as a major source of confidence that the team is getting somewhere. Since integration and testing are not left until the very end phases of a project, the team will have a better understanding of how much work they have actually completed and what the current status of the system they are building is. The amount of second guessing is reduced, because the team has results of the CI-builds. It is even possible to gather some statistics from the builds and graph them as a function of time to give the team a better understanding how the project is progressing.

Because the system is being built, deployed and automatically tested several times a day, the testers have access to an up to date system all the time. This can also be a challenge, because the system they are testing can be changing quite often and later builds might be invalidating test results of the previous build. One solution to this is to automate as many of the tests as feasibly possible. Another solution is to have a separate environment for manual testing, which is not being updated as often as the CI-build is done. This can be dangerous, because the system being tested is not the latest one anymore.

Cauldwell (2008, 22) suggests that the best way to set up a CI is to have a dedicated hardware and automate building and testing the software system. CI-builds will be performed often, usually multiple times a day. The amount of work

to build, deploy and test the system multiple times a day is so staggering that automation is the only sensible solution.

12.2 Testing Against Interfaces

In a continuous integration environment tests are being run several times a day, often against services that do not exist yet. It is tempting to skip testing in those cases or defer it until the required service is ready and can be used in testing. Doing so would lose some important feedback from the client side of a library though. According to Whittaker et al., Google has solved this problem by extensive use of testing against pre agreed interfaces instead of concrete implementations. This also speeds up development, because services can be constructed in parallel fashion (Whittaker et al. 2012, 21.) Another good effect is that pieces of software system are isolated from each other with interfaces and the library behind the interface can be switched to another implementation.

However, it is crucial that the interfaces are specified and understood when programming in this fashion. Chapter 3.2 showed an example where Mars Climate Orbiter crashed because the data interfaces were used incorrectly. Similar situation could happen if two components are developed in isolation and only unit tested against interfaces. Luckily integration tests can sometimes detects mistakes like these.

12.3 Responding to Build Breaks

One reason for using continuous integration is to detect possible problems as soon as possible, preferably within some minutes after the offending code has been commited in to the version control. An automated system will detect the changes and subsequently run a build and test it. If all the tests pass, nothing else happens; however, if the build fails, an alert is given to the team that they know to start fixing the problem.

Whittaker et al. (2012, 32) describe how most of the large projects at Google started rotating team members into the role of *"build cop"*, whose job is to respond quickly to any issues uncovered in a project's CI-build. A single responsible person tends to react faster than a team with shared responsibility. Rotating the role

naturally spreads knowledge on the builds and tools used to make them to a wider group in team.

Some CI-tools have web interface that can be used to view results of any given build and drill down in test results. While these are nice and very informative, they are not mandatory. A very basic setup where changes in source control are detected automatically and the build and test cycle are triggered is enough. Good example for such a tool is *nosy* that detects changes in disk and triggers test execution (Latornell, 2011).

12.4 Summary

Continuous integration seeks to detect mistakes in a software as early as possible by automatically integrating changes done by the developers and running the test suite against the build results. The system needs to be automated in order for it to be as fast and efficient as possible. Because tests are run automatically the team members are encouraged to commit their change as often as possible. The sooner they commit their code the faster they get feedback regarding to their changes.

Continuous integration does not usually require very expensive tools and there are many open source alternatives to choose from. If the codebase is very large, the continuous integration process will of course be slower. Then it might make sense to run only part of the test suite during the CI-build and execute the full suite during a night.

Since CI-builds are a sort of a heartbeat for a project all the developers should be concerned on the results and strive to keep the builds working. The developers should not be afraid of breaking the build now and then; it only means that the safety net was working and helped them to detect the problem before it got any further.

13 Organisational Development

13.1 Team Triad

Every well functioning team requires three main elements, which are: dedication, skills and required tools. The most important of these is the dedication or will to

perform. Tools can be handed out by upper echelon and skills can be trained by coaching or in courses, but without dedicated people the performance of the team will not be optimal.

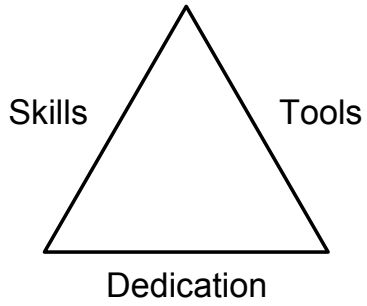

Figure 10: Team Triad

Dedication or motivation is an easy thing to destroy and hard to build. A close two-directional communication is a good start in ensuring that the developers do not feel like their opinions are not valued. Because the developers are working daily with the code and software tools they also know the best what parts of the process feel clumsy or tedious. By listening to their opinions one can gain a very valuable input and improvement ideas. This should also apply when rolling out a new process or toolchain.

13.2 Competence Development

Thomas and Hunt (2002, 38) explain how writing tests will affect the way the code is designed. If writing tests are done in parallel with development work, designers are provided with constant feedback on which classes and methods are easy to use in isolation and which have a tangled mess of dependencies all around the system. Since writing tests for loosely coupled, well designed code is easier than writing them for a really messy code, developers' design skills tend to improve almost automatically. (Thomas and Hunt 2002, 38.)

Developers who actively work on acceptance level tests or code that is being tested by acceptance level tests are exposed to how domain experts see the system and what kind of terms they use to describe its functionality. They will naturally learn to use the same terms and understand them in at least a quite similar way as the domain experts. This accumulated business knowledge can be extremely important when a domain specialist is unavailable for a reason or another.

Because the tests capture how the functions, classes and modules are intented to be used, they slowly create an encyclopedia for the developers to work with. Instead of digging inside of a module in order to understand how to use it, a developer can check the tests that are written for it. Those tests show how the module is supposed to be used and what kind of methods it supports. They can also be used to document how the module behaves in case of errors.

13.3 Easing the transition

Whittaker et al. (2012, 54-58) describe how Google wanted to make testing activities a part of every feature developers' daily work. After making it a light hearted competition between teams and providing guidance and help the progress was still slow and hard. Eventually they had to scale things down somewhat and start with really simple tasks, like setting up a CI build and classifying their tests to small, medium and large. The goal was to get the people started with something easy that would provide quick gratification and get them hooked to continue further. (Whittaker et al. 2012, 54-58.)

Karten (2009, 22) points out very important lesson: *"How you implement a change this time will affect how people respond to future changes."* Mistakes made now will have negative effect in the future. Therefore it is important to give everybody a chance to participate to the roll out of new tools or practices. Realistically speaking there is not enough time or space for everybody to participate, but simply giving them a chance is often good enough. The people who are more inclined to participate and voice their opinion will do so and the more silent people will feel that their opinions are valued too. Karten has come to similar conclusion and identified that *"even a minimal sense of control can go a long way toward easing the stress people feel"* (Karten, 2009, 38).

When communicating plans or upcoming changes it is good to remember that the way you communicate the upcoming change affects the duration and intensity of the chaos that the change will bring (Karten, 2009, 56). Too much information too early might lead to unnecessary speculations and the plans might still change multiple times before they are actually implemented. Too little information or giving information too late is not a good thing either because the change might come as a surprise and have too negative effect on daily work. Finding a good compromise

between these two extremes depends quite a deal about the people involved and the organisational culture. The skills and experience of the superiors will most likely be useful in situations like this.

13.4 Summary

Even when the change is about tools or technology the people affected are one of the most important factors. Without motivated people no tool or process will work well. Open communication and trust are the corner stones that a succesfull change is built upon. Sometimes technically oriented people might forget that the people - not the tool - are important. They also might have difficulties in communicating their plans well to all respective parties. In such cases the superiors could step in and offer their help. The team, group and department leaders could have useful experience that could be utilised.

14 Implementation in the Host Company

14.1 Motivation

Chapters from 14.3 to 14.13 will summarize implementation phase of the thesis in the host company. The chapters cover roughly the same subjects as the theoretical part and show how the theory was put into practice.

14.2 Overview of the System

The software system that was the target of the testing in the present thesis is rather old, roughly around 20 years and it is still under active development. It has 2.5 million lines of code and can be tailored to customer needs very well. This in turn means that the code is rather complex in some places. Since the domain of the system is finances, there are some very specific requirements mandated by the Finnish legislation and requirements for error free operation of the system are strict.

The host company was using TeamCity for the build management. It is a Java based build management and continuous integration system which, among other things, features automatic build triggering based on commits in version control system, notifications and code coverage tools (Melymuka, 2012, 9-10).

For test case management the host company was using SpiraTest. The tool offers ability to manage requirements, tests, bugs and issues in a very comprehensive suite (Inflectra, 2013).

14.3 Test Execution

The tests were grouped by the technology they were built upon and their execution speed. Very quickly executed unit tests are run several times a day as a part of continuous integration build (see chapter 12 for more information). More time consuming integration tests are run during night, when the server load is low. A subset of integration tests was selected to be run during day as a smoke test that can be used to verify general state of the software system.

In NUnit, it is possible to assign a category for tests, which can be used to select only a subset of them to be executed. This was used to label some of the integration tests for smoke testing. The problem with this approach is that while TeamCity is able to filter tests based on their categories, it is possible to assign only a single category for each test. The same test can not be labelled as belonging to a certain customer and as a smoke test easily. One possible solution is careful management of categories and compounding different tags together, e.g. *"customer_A"* and *"smoke"* together becomes *"customerA_smoke"*. This will get unwieldy as the amount of tags grow and complicate management of test cases.

Another option is to separate test cases to different dll files based on their usage and run tests only for a specific set of them. This will run into the same problems as using categories, since a single test case naturally can not exist in multiple dll files. Also the management of builds becomes harder, since the tests can not be picked up easily by wildcard filters anymore.

The current solution is to assign tests to a category based on information whether they are common or customer specific. Few tests were picked for smoke testing and because they are not customer specific there is no clash in categories. However, In the future the situation might change as the amount of the tests keeps growing.

14.4 Unit Tests

Writing unit tests for an old and complicated legacy system is not an easy task and it took several tries before a suitable method of working was found. Especially when existing code was being modified and extended, writing unit tests was a real challenge, because of the *"test smells"* which were mentioned in chapter 5.2. Usually the best approach was to first write some integration tests to cover the feature in question and use them to verify the correctness of the functionality, while breaking the code into smaller components. Those smaller components could then in turn be tested with unit tests to verify that they are working as expected. After that the development of new features could start for that specific component or subsystem. This of course was a rather slow approach and it was hard to justify the time spent in the beginning, especially because most of the bugs were revealed by integration or end to end tests.

NUnit and NSubstitute were testing frameworks used. While NUnit can be used to define and run tests, NSubstitute can be used to substitute dependencies in unit tests. Both tools are in a widespread use and it is easy to find examples and studies regarding to them.

14.5 Integration Tests

The case system had integration test platform that had been modelled after examples given by Freeman and Pryce (2010). The platform was expanded with more builders and matchers were introduced to help verification of results. A similar platform was created for unit testing.

The integration tests were built using NUnit. The tool was chosen, because it is used widely for testing and offers the basic functionality needed for running the tests. Because of the widespread usage of xUnit tools there are plenty of other tools to choose from for reporting and test management. This helped to integrate integration tests later with TeamCity and SpiraTest.

Figure 11 shows on a conceptual level the relation of client-side classes and integration tests. From the point of view of the application server the integration tests behave just like any other client application. They log in to the application

server and start executing the business logic accessible to them via web services. While this ensures that the tests resemble the actual usage patterns of the system as closely as possible it also means that the tests are rather slow. Communication over a network adds latency to the tests and possible point of failure in the form of network errors.

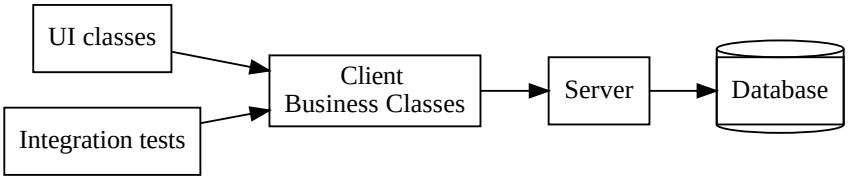

Figure 11: Integration tests

The advantage of reusing client side business classes is that it is easy to communicate with the application server and all the changes done on the business classes are readily available for the integration tests. Since duplicating logic present on UI layer in the test classes is tedious, this also encourages the developers to move the logic away from the UI layer and into the client or server side business classes.

14.6 End to End Tests

While the choice of the platform for unit tests was easy, especially end to end tests proved to be a subject of heated debate. Because end to end tests are potentially used in communication with stakeholders like product owners, domain specialists and customers, the clarity of tests was an important matter. Various tools were investigated, FitNesse and SpecFlow being the prime candidates of choice.

FitNesse is a software testing tool with a focus on collaboration between stakeholders. It is built on top of a wiki and allows customers, testers and programmers to learn what their software should do and compare it to what it actually does (FitNesse - One Minute Description.)

SpecFlow is a pragmatic testing framework for behaviour driven development and acceptance tests. It aims to bridge the communication gap between domain experts and developers by providing ability to write human readable behaviour specifications and examples that can be executed as tests (SpecFlow - Pragmatic BDD for .Net.) SpecFlow was tested by a few developers, the author of the present thesis being one of them.

In the end, the choice was made to use FitNesse for end to end tests. This decision was made because there already was a substantial amount of tests written with it and expanding its use would be easy. SpecFlow on the other hand would have been a completely new tool. A very attractive feature of FitNesse was the internal wiki-system that is used to run tests. While the tool has not been taken into use by other than developers, in the future it is possible to publish all the test cases, their descriptions and explanations on an internal website. This would create a more transparent way of working regarding the tests and their results.

FitNesse arranges test data in a tabular format that makes it easy to test data rich application (Koudelia, 2012, 72). This approach is very suitable because the system under test in the case of the present thesis is a financial system. The system required for testing is slightly more complex, since FitNesse requires own server to run the wiki. Setting up one is rather simple operation though, even on a developer's machine.

14.7 Matcher Library for Assertions

As mentioned in chapter 7.4, writing clear assertions will make it easier to understand what is being tested. For this purpose NHamcrest was chosen for evaluation. According to Hay, NHamcrest is a C# port of the Java version of Hamcrest. It offers a good set of matcher classes and a possibility to define your own. (Hay 2010.)

As of writing the present thesis, there has not been a real compelling reason to use NHamcrest in testing. While it offers the ability to write clean and easy to understand assertions, the assertions needed in testing in this case did not require those capabilities. The teams felt that using built in assertions of NUnit and some custom written ones they could achieve the same end results without resorting to a third party library. The ability to have nicely formatted, close to a natural language error messages is probably not so important when only the developers are investigating the error reports.

14.8 Domain-Specific Language for Testing

In order to hide some of the less interesting technical details and make tests more readable, domain-specific languages were investigated. Main focus for the DSL was

in integration tests, because they would benefit most from the detail hiding. The amount of unit tests was smaller compared to the integration tests and one of the driving forces in adopting FitNesse was the ability to display large amounts of data efficiently.

The framework that used writing integration tests was implemented with a fluent API, essentially it is an internal domain-specific language (refer chapter 8.2.1 for more information). The goal has been to make writing expressive tests as easy as possible and let developers concentrate on business properties of objects, instead of their technical details. Partly this has been successful, especially in those tests that use the more mature parts of the framework. More work is needed though, since the framework covers only a fraction of the needed cases. Basis of DSL used is in construction builders (Fowler and Parsons, 2011, 179) which are controlled with method chaining as defined by Fowler and Parsons (2011, 373).

Adding tests to already existing system was somewhat tedious from the point of view of the DSL too. In the beginning there were no builder, matchers or any other helpers to rely on, so almost each and every integration test required writing some helpers. This made developing tests both slow and tedious. The developers felt that it required writing too many classes to get even a simple test case done. However, as explained in chapter 8.3, it is possible to use an internal domain-specific language as a springboard for writing an external domain-specific language. For example, the builders could be reused when writing tests with FitNesse.

14.9 Reporting

14.9.1 Reporting Test Results

Tests are not very useful if there are no comprehensive reports showing what tests were executed, against which system they were executed and what the results were. At the very start, test execution was started manually and the results were not stored anywhere. This basic setup served as a sanity check for developers, who could get a general feeling if certain limited amount of features were working correctly.

The next step was to automate the build process and at the same time automate the testing process. The build process was automated using TeamCity. TeamCity can store various data from builds, including test results. This allowed developers to

see test results of any previous run, graph them and compare different builds. At that point it was possible to see on a coarse level how quality of features was developing over time.

Eventually TeamCity was integrated with SpiraTest. Results from various automated test suites were automatically imported to SpiraTest for later review and analysis. This allowed everyone to view the test results, drill down into data and view graphs showing test results as function of time. Because SpiraTest was also used to store results of manual testing, it offered one stop shop for all things related to testing.

14.9.2 Test Coverage Reports

One of the metrics closely associated with test reporting is code coverage. It is used to report how big a portion of the software is being covered by tests. As pointed out by Kaner (1996, 7-13), there are many ways of defining and calculating coverage, each yielding somewhat different focus on the software testing. Since the host company is using TeamCity for build management, dotCover was natural choice for coverage analysis. Both tools are developed by the same company and integration between them is good. DotCover can be used to report the percentage of lines being covered by the tests and also generate a report highlighting which lines were executed and which were not.

TeamCity is shipped with console version of dotCover, which enables coverage analysis out of the box. It was configured to collect statistics regarding to the execution of unit tests. While the coverage analysis of unit tests was easy, integration and end to end tests proved to be somewhat more difficult. Because dotCover needs to be running on the machine that is executing the tested code, analysing server side code means deploying and running dotCover on the server. This approach has the advantage that it is possible to gather coverage metrics for manually executed test cases too and combine them with results from automated test cases. This gives stakeholders an overview of what parts of the software were used during testing.

Collecting coverage data during test execution has a negative impact on performance. Typically it seems to be 3 to 4 times slower to run tests with the coverage reporting turned on than without it. Therefore the coverage reporting was turned off when running CI-builds and only collected during nightly builds. The

impact to the total build time is somewhat smaller, since the build process consists of multiple steps in addition to running the test cases. Based on test builds with CI configuration the build times with coverage reporting on are roughly 60% longer than without it. In the nightly builds where a deployment is done the difference is smaller, around 25%.

14.10 Dependency Injection

Dependency injection (discussed in more detail in chapter 9.3) was not in wide spread use in the host company before winter 2011-2012. Originally developers started experimenting with it, using poor man's injection where no inversion of control container is needed. After all, DI-container is not a hard requirement for dependency injection (Seemann, 2012, 197). This was a fast and straight way forward and enabled immediate returns. The style of injection was chosen to be constructor injection, because it made the required dependencies clear and removed temporal coupling. As expected, static coupling between components in the software system decreased as a result of dependency injection. Transition to a new way of coding was not without problems though. It was often noted that loosely coupled software based on copious use of interfaces made it harder to understand how the code works without debugging it.

Originally dependencies were substituted by using statically defined test components, but soon the focus switched over to dynamic substitution and mock-libraries. The first tool taken into use was moq, but after a trial period NSubstitute was deemed a more promising approach. NSubstitute so far has not been lacking any really significant features.

As the time progressed and components being injected started getting more complex, it was noted that a better solution was needed. At this point few developers started experimenting with Unity, which is an open source IOC-container and created couple of demos to showcase its usage. Even when the concept is really simply, it required considerable effort to design, develop and test an approach that would play well with the existing code. Because the software system in question is old and large, it will most likely never be rewritten to use IOC-container everywhere. Instead of that, the old and the new architecture will have to live side-by-side in a way that it is possible for the parts to interact with each other.

14.10.1 In-house Service Locator

The software system was using ObjectFactory, a service locator written in-house, to instantiate objects. Listing 21 shows an example where a Customer object is instantiated and loaded from the database. The system is type safe and offers a possibility to define custom version of any component and configure it in use. In such a case, instead of returning an instance of Customer, the ObjectFactory could return instance of Customer_Custom class.

```
Dim customers = ObjectFactory.CreateInstance(Of Customers)
customers.LoadAllCustomers()
```

Listing 21: Instantiating object with ObjectFactory

The system was not without drawbacks though. The major one was the lack of ability to create an instance of an object based on an interface, which resulted in hard dependencies between concrete classes. Eventually this was addressed by adding Unity into the system and integrating it with the ObjectFactory. The factory was still used to create instances as before, but if a caller instructed it to create an interface, the execution was forwarded to Unity. Listing 22 shows an example how new ObjectFactory could be used.

```
Dim repository = ObjectFactory.CreateInstance(Of ICustomerRepository)
Dim customers = repository.LoadAllCustomers()
```

Listing 22: Integrated ObjectFactory and Unity

This made it easier to decouple components from each other, because the configuration of the system was not based on concrete classes anymore, but on interfaces. Developers still have to keep in mind that while earlier every component could create dependencies it needed, now those dependencies should be injected from outside.

14.10.2 Tackling Dependencies

New ObjectFactory with integration to Unity of course was not a solve-it-all solution to handling dependencies. Because the old and the new architecture have to live side by side for undefined time, it was not possible to take a pure approach on dependency injection. In the pure approach, there would be only one composition

root, where all the needed objects are resolved against IOC-container configuration. After that point, there would be no calls to IOC-container at all. This was not possible, because the old and new code had to have a way to access functionality of the other. Instead of a single composition root, calls to ObjectFactory (and to Unity in turn) could be made from anywhere from the software.

In essence, if the developers were not paying close attention, the situation presented in Figure 12 could happen. The system presented in the Figure is hypothetical; however, it illustrates the problem well. JobManager is an object that sorts objects by delegating the task to Sorter object. The sorter object in turn uses various algorithms to sort objects (bubble sort in this example).

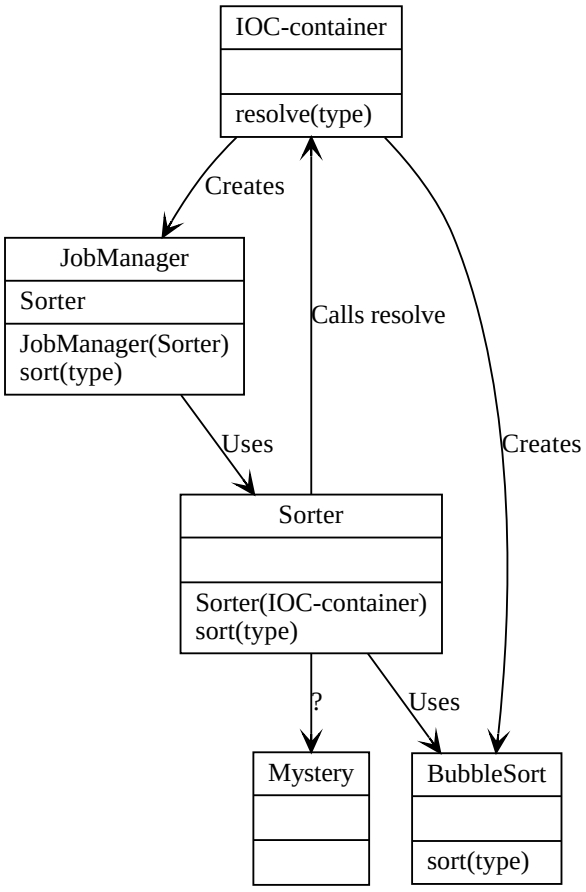

Figure 12: Mishandled dependencies with IOC-container

On the surface everything looks to be in order: instance of JobManager is created by IOC-container and instance of Sorter is supplied to it via constructor. However, Sorter has IOC-container as a dependency, because it needs IOC-container to create an instance of BubbleSort class. This means that even if IOC-container is supplied through constructor to Sorter object, the caller has no way of knowing that the

Sorter needs instances of BubbleSort and Mystery objects in order to work properly. Even more alarming is the fact that in order to unit test Sorter object, one has to construct an instance or a mock of IOC-container and configure it to return correct instances. And the only way to know this is to either trial and error or reading the code of Sorter class. The situation is even worse if an instance of the Mystery object is created directly by calling the constructor. This way there is no easy way to use a substitute instead of a production version.

In order to manage the situation, a set of guidelines was developed. All the new code that is written, should compose as much of the object graph as possible in a private composition root. Composition root could be placed in a suitable seam, like beginning of a web method, inside of a facade, or other suitable location. After that point, calls to ObjectFactory and Unity would be avoided at all costs. This would ensure loosely coupled software and testability. Legacy code was harder beast to tame. Developers could refactor code, provided that it was covered by integration tests, to accomodate testing where it made sense. Where refactoring was too difficult or otherwise impractical, old code could create a private composition root when making a call to code written with new architecture. Calls from old code to old code were left alone. This was used to contain tangled dependencies and keep them from spreading from old side to the new side.

14.11 Continuous Integration

The host company was already using continuous integration (see chapter 12.1) for all teams. The solution chosen for this was TeamCity. Depending on the team, the build was scheduled to run 15 minutes after the latest commit and build either full or part of the software. This was improved by adding unit tests as a part of CI-build using TeamCity's NUnit test runner. The initial solution was further improved by decoupling building of client and server software. In this model, if only client-side code is changed, only client is built and tested. If there are changes both on client and server, they are built and tested in parallel, thus speeding up the execution.

To facilitate quick response on build breaks several methods are used. TeamCity has a built in capability to send email to people who have made changes that might have caused a build break. Similar information is available from a small Windows tray notifier program that shows status of selected builds and will pop-up a

notification when a build succeeds or fails. Melymuka (2012, 63) lists other options for notification, like RSS-feeds and Jabber. They were considered, but not taken into use at this point.

The latests method of staying informed on build status is an internal web page called *"radiator"*. This page displays each selected project as a green or red box, depending on the status of the latest build. From the start the radiator was available for anyone via a web page. Later on, a spare computer and screen were setup to continuously show the build status so that anyone stepping into office would see it immediately.

The role of build cop (mentioned in chapter 12.3) was taken into use in one team. The system seemed to work rather well in the beginning and responses to build breaks were quick. Slowly it fell from favour though and practice was discontinued. However, this did not mean that builds were left in a broken state for days. Even when there was no specifically assigned role that was responsible for broken builds, team members actively started investigating why builds broke and what was needed to fix them. At this point the team understood the value of tests and the feedback that they provide and wanted to keep that feedback system working.

14.12 Verification of Customer Test Environment

Many customers of the commissioner have one or more test environments that they use to test the software before deploying it into production. These environments can be as complex as the production ones, with multiple external systems which they have to integrate with. Traditionally verifying deployment done to environment like this has been performed manually and the content of the verification has varied from person to person.

Because integration tests verify large parts of the system, from the client to the database, they are well suited for verifying that the test environment is in working state. Some of the external systems are not available to the commissioner at all and they have to rely on interface specification and various test programs in order to be able to develop the software, so testing against them in the customer's test environment is an extremely valuable opportunity.

14.13 Training

Testing of a legacy system is probably the hardest possible way to start with automated testing. The software system in question is old, large and complex. End to end and integration tests were relatively easy to apply, but unit tests were really difficult. To get everybody in the same line regarding to what kind of software design and code is required for writing a testable system, series of general software design trainings were held. In these trainings, developers were shown certain patterns and methods that allow them to write code that is easier to maintain and test. The trainings were relatively short, concentrating only on few topics at the time, so that developers would have time to properly digest the material before the next training.

Training for testing was arranged in a similar way. Few short sessions where held where basic principles of automated testing were presented. Most of the training was conducted in hands-on approach, where more experienced developers provided help and guidance for others.

The teams share a development blog, where anyone can post about development related matters. While testing practices evolved and new information was obtained regarding to the problem domain, developers were encouraged to share their experiences, ideas, tips and tricks with others via a blog. This made the information readily available to everyone and at the same time it was collected and preserved in a single location. In the spring 2013 the blog was enhanced by adding a forum where the developers could discuss with each other in a transparent manner. The information would be readily available and the communication would not be bound to a specific time or place.

Between autumn 2012 and sprint 2013 teams arranged test automation camps, where they allocated half a day for whole team to work on test automation. They were given free hands by leaders of the department to arrange it in a way they liked and work on the matters that they deemed most important or interesting. The only constraint was that the teams had to present their results to the management. Feedback from the camps was very positive and many developers said that they were having a really great time and felt productive.

15 Surveys

15.1 Overview of Surveys

During the summer of 2012 developers were presented with the survey that is shown in Appendix 1. The data collected from it was collated and is presented in Appendix 2. The original answers for the survey are held by the author of the present thesis. Chapter 15.2 takes a closer look at the data and some of the deductions that can be drawn from it. Chapter 15.3 does the same, but focuses on the second survey, which was done on January 2013.

In between of the surveys the work described in chapter 14 was carried out. The aim of the surveys was to measure the effect of the aforementioned work would have. Chapter 15.4 contains an analysis of the differences between results of the first and the second survey.

15.2 The First Survey

Table 1 shows some key figures on qualitative variables of the first survey. The survey was sent to 33 participants and 17 answers were received.

Table 1: Statistics on quantitative variables of first survey

Variable	Minimum	Maximum	Arithmetic Mean	Standard deviation	Median
verify_local	2	5	2.823529	0.808957	3
verify_global	2	6	3.882353	0.992620	4
understand_local	2	4	2.882353	0.600245	3
understand_global	1	5	3.705882	0.919559	4
returning_bugs	2	5	3.176471	0.882843	3
bug_local	2	5	3.000000	0.866025	3
bug_global	2	4	3.000000	0.866025	3

As expected, both understanding how the software works in the broader scale and verifying that changes done by a developer have not broken any functionality in broader scale is harder than understanding and verifying only the functionality that

was changed. This was expected, since the software system in question is large and very complex. 50% of the developers who answered the first survey cited lack of business knowledge as one of the major challenges in testing in general. The data is presented in Figure 13.

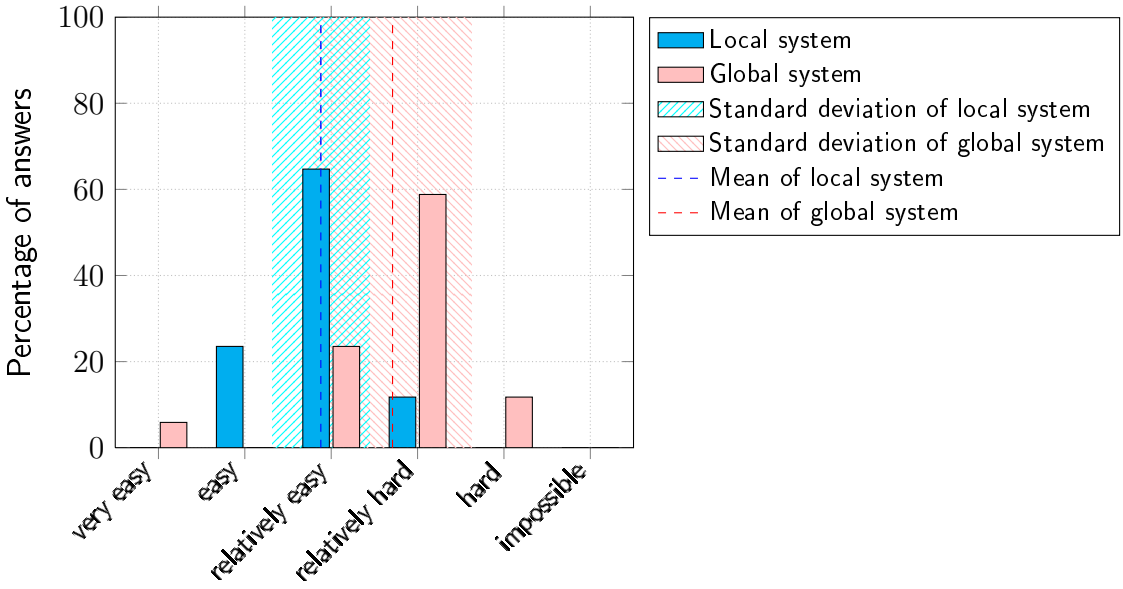

Figure 13: Ease of understanding the system

Figure 14 presents the data that shows how verification of changed functionality was perceived being easier than verification of the system in general.

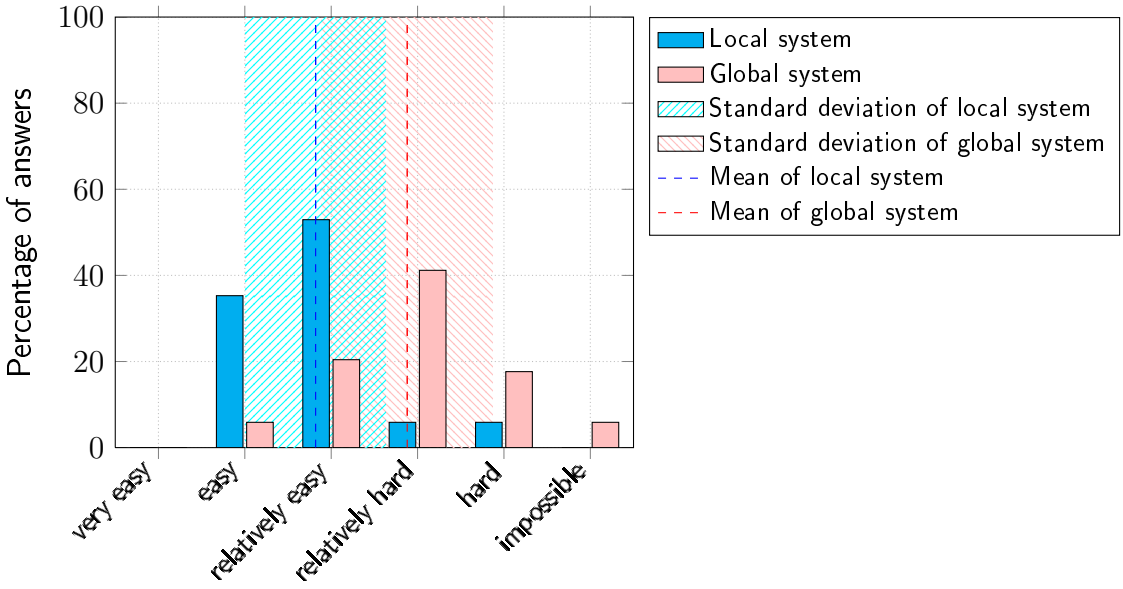

Figure 14: Ease of verification of functionality

Figure 15 shows how often changes done by developers caused defects in a changed functionality and in a completely unrelated functionality. The results are somewhat

surprising, because almost an equal amount of developers selected *"Very rarely"*, *"Relatively rarely"* and *"Relatively often"* to the question *"How often the changes you make (including database changes) create unexpected problems, in somewhere completely unrelated part of software (so called house of cards effect)"*. Answers to the *"How often the changes you make (including database changes) create unexpected problems, in the functionality you changed"* however roughly followed standard distribution and were clustered toward *"Very rarely"* and *"Relatively rarely"*.

Based on the answers given it seems that the developers are as likely to cause defects in the local part of the software as on the global part. The original expectation was that the local software would be easier to work with than the global one.

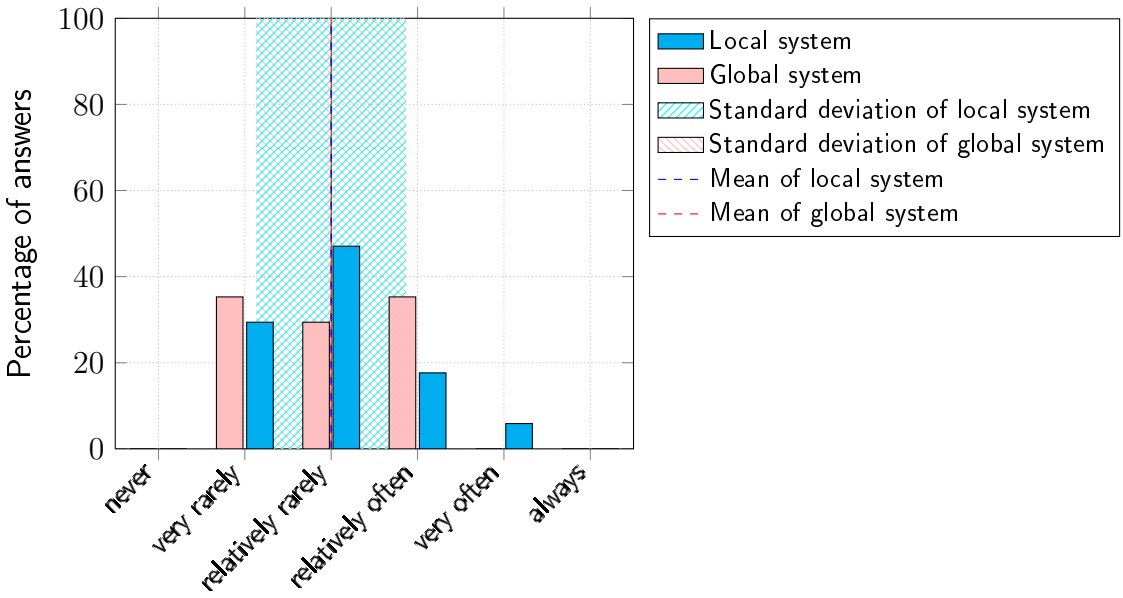

Figure 15: Defects caused by changes

Figure 16 shows how the majority of developers have problems with regression relatively rarely. The standard distribution is relatively large though, being 0.88. Only a very few developers had problems with regression very often.

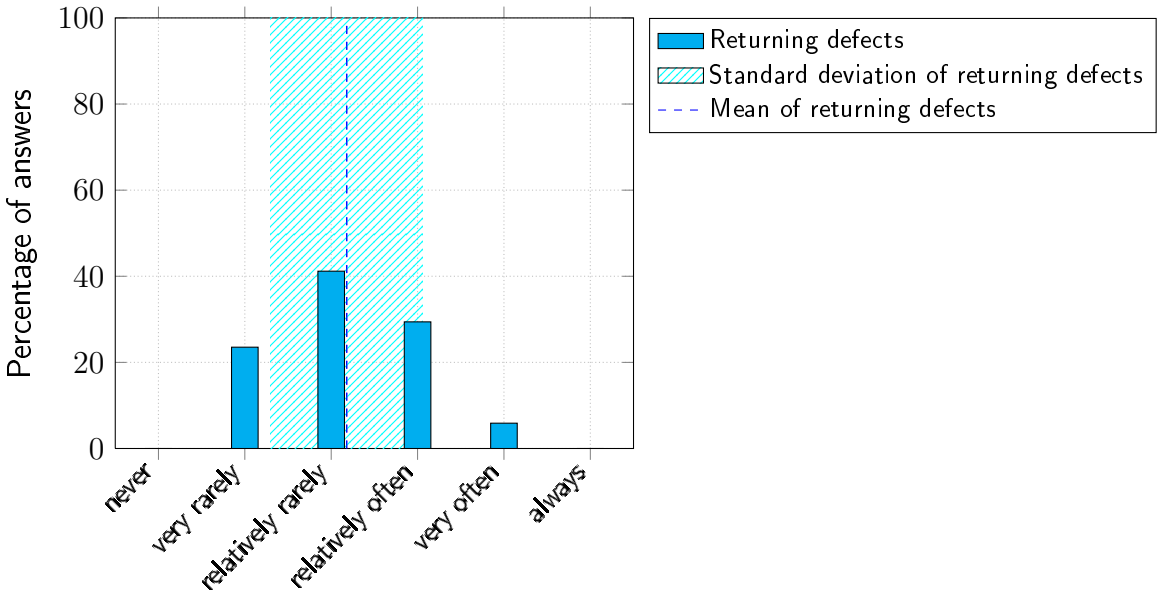

Figure 16: Returning defects

15.3 The Second Survey

Table 2 shows some key figures on qualitative variables of the second survey. The survey was sent to 33 participants and 16 answers were received.

Table 2: Statistics on quantitative variables of the second survey

Variable	Minimum	Maximum	Arithmetic Mean	Standard deviation	Median
verify_local	2	5	2.75	0.856349	3
verify_global	2	6	3.8125	1.167262	4
understand_local	2	4	2.875	0.806226	3
understand_global	2	5	3.25	0.930949	3
returning_bugs	2	3	2.8125	0.403113	3
bug_local	2	4	3.0	0.516398	3
bug_global	1	4	2.8125	0.75	3
fixing	1	3	1.733333	0.593617	2
quality	1	4	1.875	0.957427	2
debugging	1	4	1.8125	0.910586	2

Figure 17 shows how understanding the local system is easier than understanding the system as a whole. The standard deviation in the case of the local system is

slightly smaller than in the case of the global system (0.81 as opposed of 0.93). The arithmetic mean in both cases is at *"relatively easy"*.

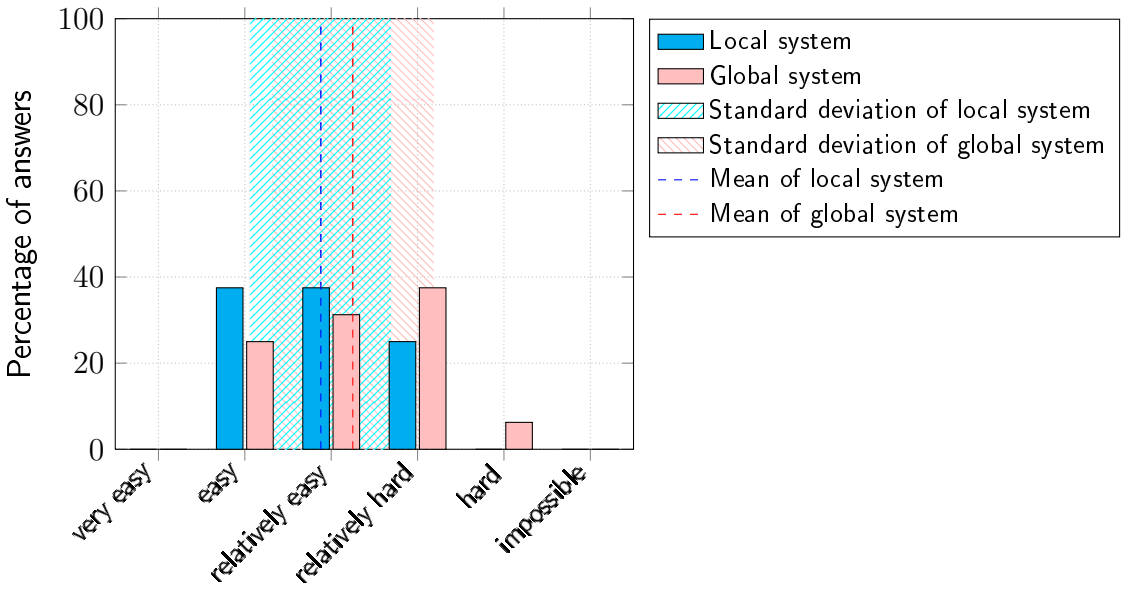

Figure 17: Ease of understanding the system

Figure 18 highlights how big a difference there is in verifying the changes in the local context compared to the global context. Some of the developers even feel that they are unable to verify that their changes did not break anything in system's global scale. This is a rather alarming find, because the financial system in question has very strict requirements for being error free.

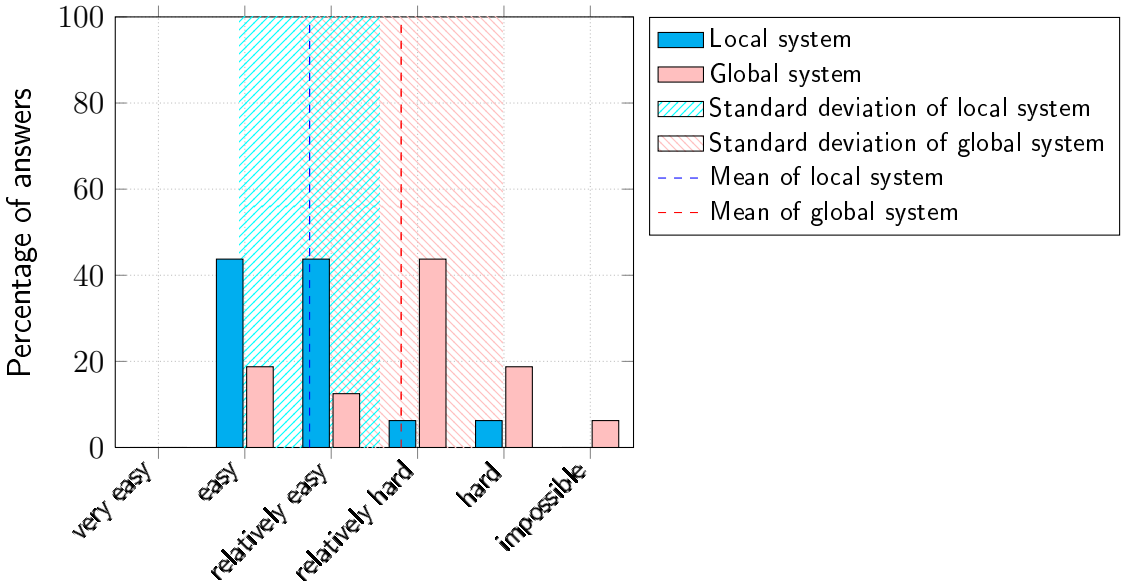

Figure 18: Ease of verification of functionality

The developers feel that changes they do cause defects in the global context of the system more often than in the local context as shown in Figure 19. While the arithmetic mean in both cases are close to each other, the standard distribution has larger differences. The standard distribution is 0.52 in the case of local context and 0.75 in the case of the global one. The difference which is quite significant can be explained by the complex system where it is not always easy to understand all the effects of a single change.

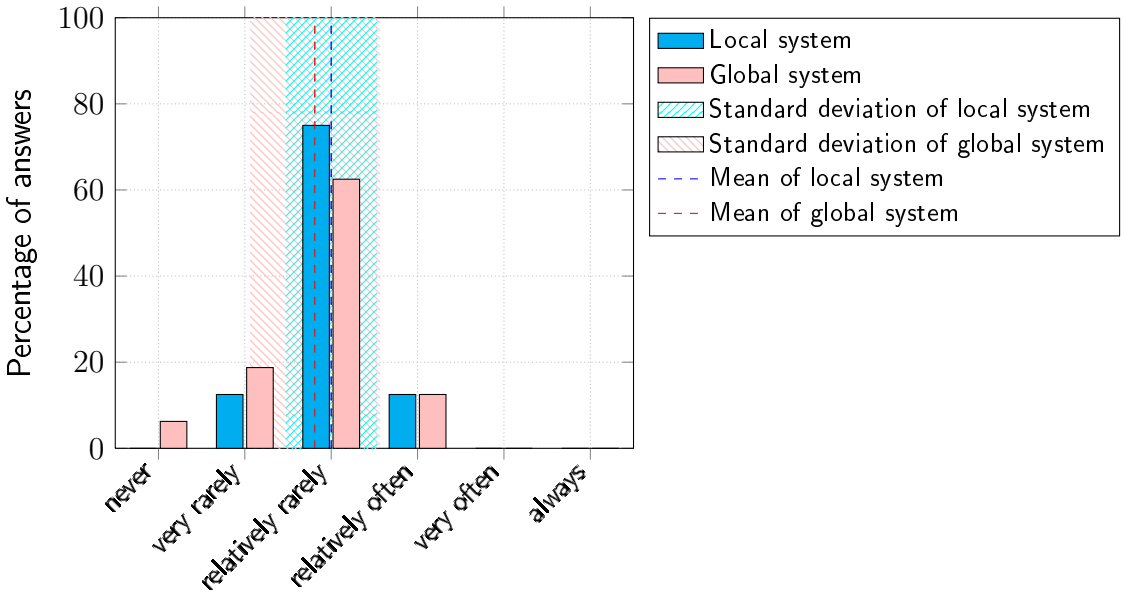

Figure 19: Defects caused by changes

In the second survey the developers felt that they have problems with regression relatively rarely. The Figure 20 shows how the answers are clustered around *"very rarely"* and *"relatively rarely"*. When comparing to the earlier graphs, it can be concluded that while the verification of the system as a whole is relatively hard, the defects that get introduced are relative rarely returning ones.

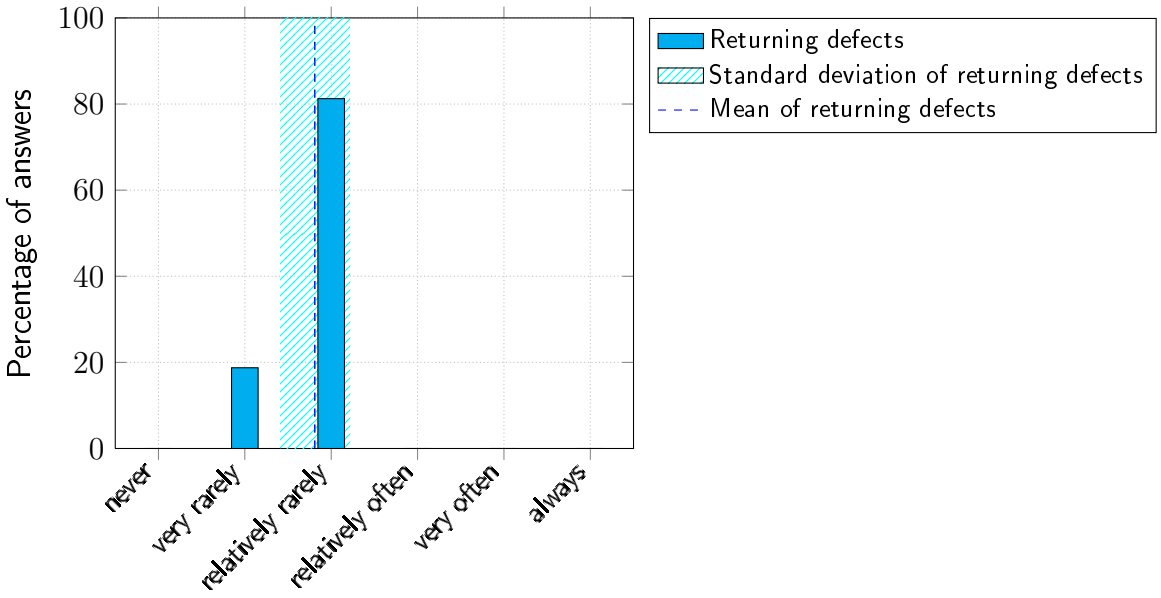

Figure 20: Returning defects

The second survey had some additional statements that were used to gauge how helpful the developers think automated testing done by them is. The statements were *"When fixing a problem in the system, I find it useful to write tests to verify my fix"*, *"Tests help me to product higher quality code"*, *"Tests are useful while debugging, when I find a fault"* and results are shown in Figure 21. The mean average is at *"agree"* in all cases; however, there is quite a large distribution from *"strongly agree"* to *"disagree"*.

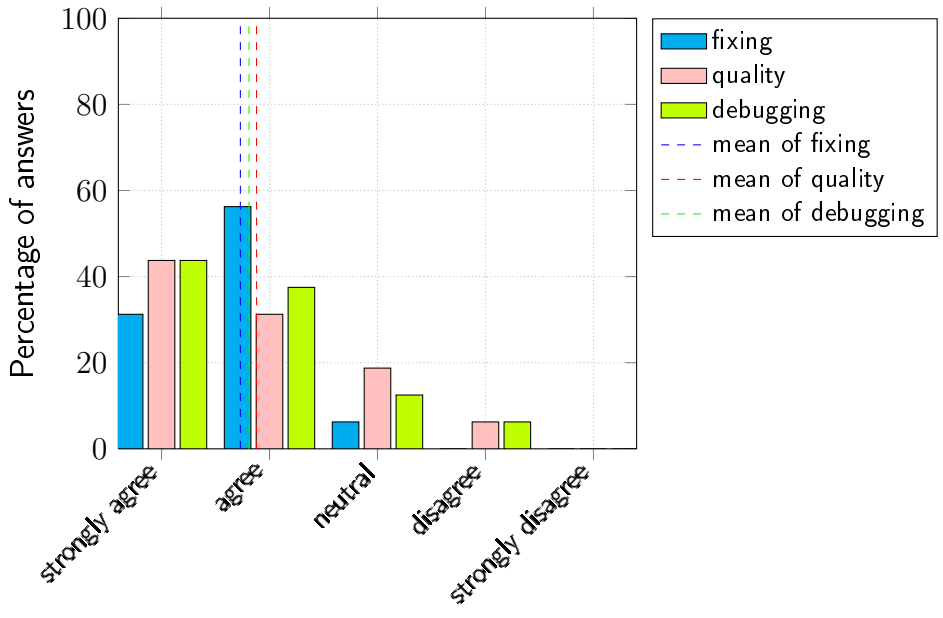

Figure 21: Usefulness of the tests

The standard deviation in the statement *"Tests help me to product higher quality code"* was 0.96 and it was the largest. Over 70% of the developers felt that automated tests helped them to produce better quality code; however, few developers disagreed with this. It is unclear if the reason was that not all developers are actively writing automated tests or if writing automated tests do not help some of the developers to produce higher quality code.

The next largest standard deviation was in the statement *"Tests are useful while debugging, when I find a fault"* where it was 0.91. Around 80% of the developers felt that tests are helpful when they are debugging a fault. Small portion of the developers disagreed with this statement and felt that the tests did not help them in debugging a fault.

The smallest standard deviation was in the statement *"When fixing a problem in the system, I find it useful to write tests to verify my fix"* where it was 0.59. Over 90% of the developers agreed that writing automated tests to verify a fix is useful. Nobody disagreed and only a small percentage had a neutral stance towards this. This indicates that the developers clearly see the usefulness of the automated testing in verifying their own work.

15.4 Analysis of Differences

Figure 22 shows answers of both the first and the second survey on question *"How hard is it for you to see how classes and methods work?"*. Answers are clustered around *"easy"*, *"relatively easy"* and *"relatively hard"*. In the first survey, answer *"relatively easy"* dominated, while in the second survey answers are more evenly distributed. Somewhat surprisingly while *"easy"* got more answers in the second survey, so did the *"relatively hard"*.

Analysing some of the qualitative variables of the results shows that the arithmetic mean has stayed constant between surveys. It has a value of 2.9, which falls between *"easy"* and *"relatively easy"*, while being very close to *"relatively easy"*. Standard deviation however rose from 0.6 to 0.8. This would indicate that while in general there was no shift in perceived difficulty of understanding the local system, the deviation between developers grew larger.

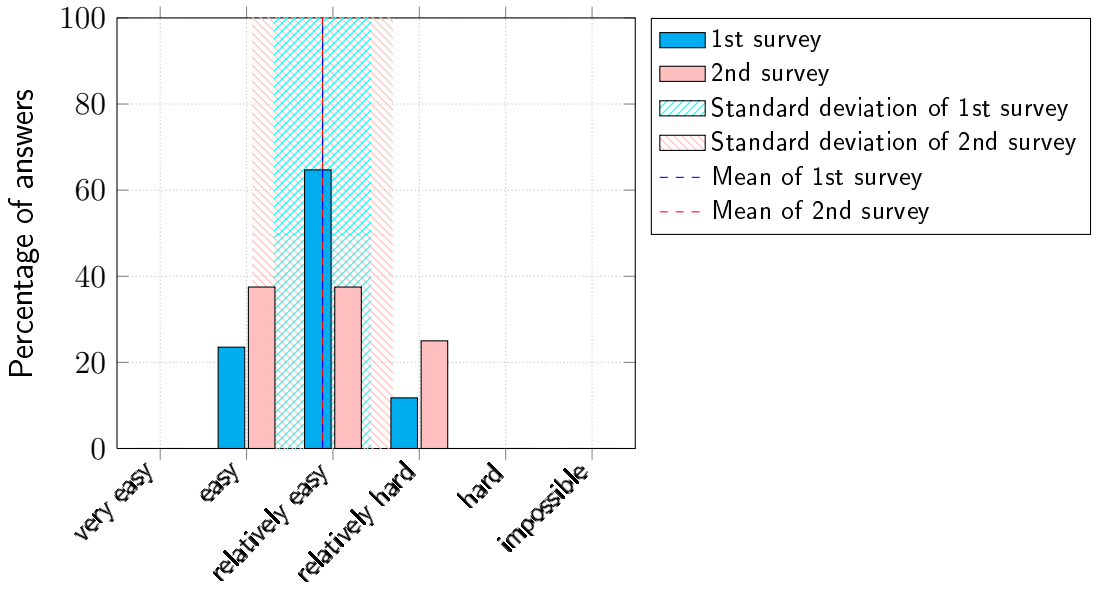

Figure 22: Difference in understanding local system

Figure 23 shows answers to the statement *"How hard is it for you to see how components work together"*. There are answers to categories *"very easy"*, *"easy"*, *"relatively easy"*, *"relatively hard"* and *"hard"*. Only *"impossible"* got no answers at all.

The standard deviation between the first and the second survey stayed at the value of 0.9, while the arithmetic mean fell from 3.7 to 3.2. While both values fall into *"relatively easy"*, the answers in the first survey are closer to *"relatively hard"*, while the answers of the second survey are closer to *"easy"*. This would indicate that in general, developers found it easier to understand how the system in general works in the second survey.

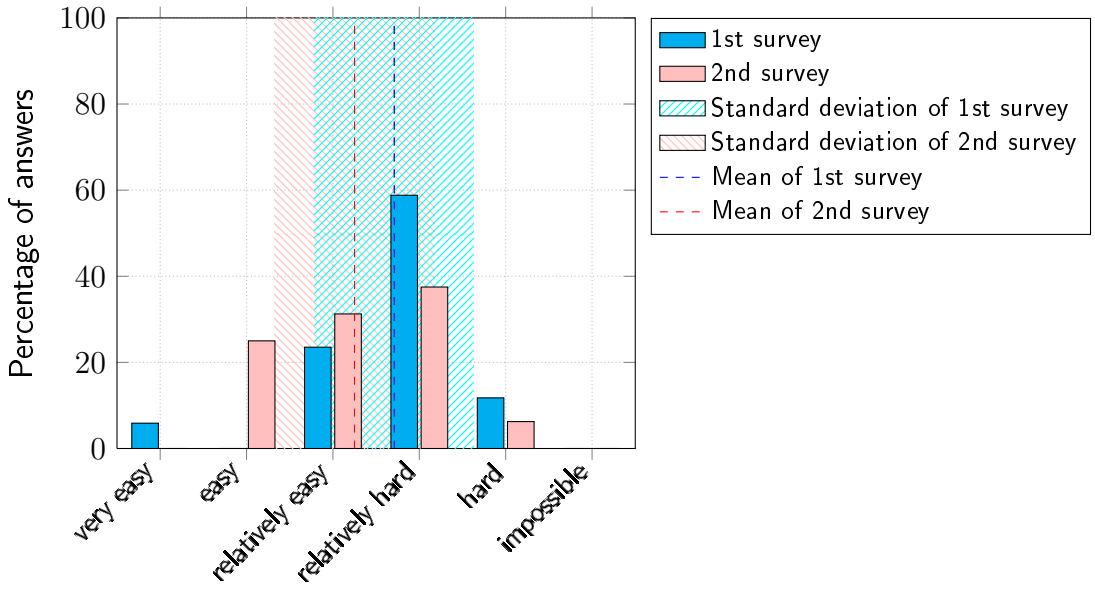

Figure 23: Difference in understanding global system

Figure 24 shows answers to *"How easy it is for you to verify the changes related to other parts of the functionality?"*. Answers in both surveys range from *"easy"* to *"very hard"*. Key figures in both surveys are almost the same, with the arithmetic mean being 2.8 in both and the standard deviation being 0.8 in the first survey and 0.9 in the second survey. There does not seem to be any notable difference between the results of the first and the second surveys.

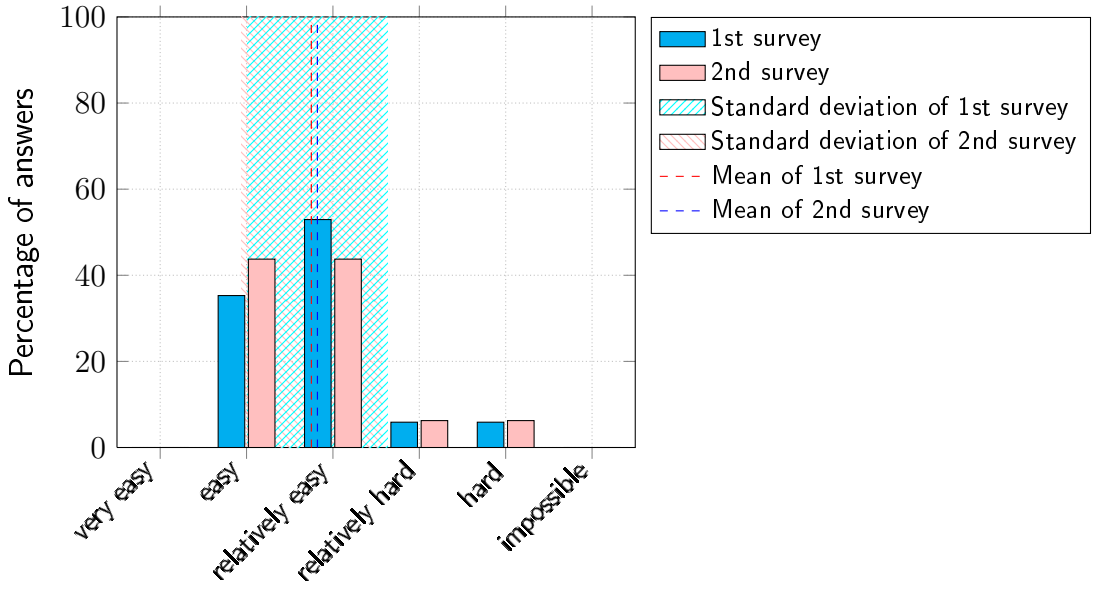

Figure 24: Difference in ease of verification of local changes

Figure 25 presents the results of both surveys to the question *"How easy it is for you to verify the changes related to rest of the system?"* While the arithmetic mean of

3.81 in the second survey is slightly better than 3.88 in the first one, the difference is not statistically notable. Somewhat surprising is the fact that the standard deviation grew from 0.99 to 1.17. This would mean that even when the developers in general felt more confident that they can verify the changes in related to the rest of the system, the difference between developers grew somewhat.

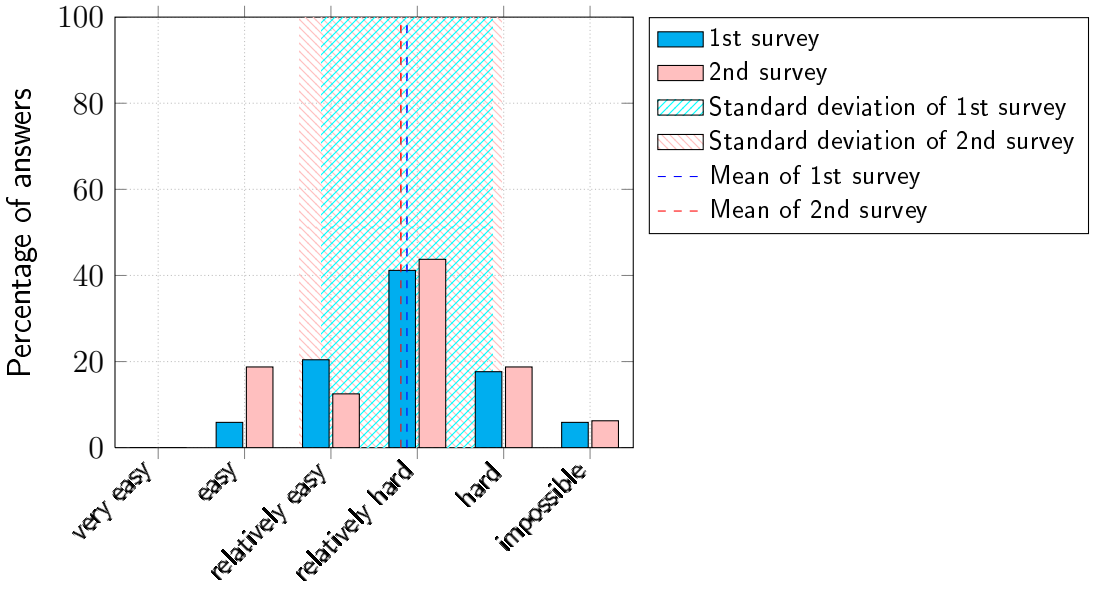

Figure 25: Difference in ease of verification of global changes

The result is different than what would be expected, if there is a global safety net, that is provided by a comprehensive suite of automated tests. One possible reason for the results is that the automation effort drew the team's attention to the fact that verifying changes in the global context is both difficult and not suitably covered by the tests. This in turn may have caused them to doubt their current ability to verify changes in global context and produced the given results.

Figure 26 shows comparison between the first and the second survey on question *"How often the changes you make create (including database changes) unexpected problems in the functionality you changed?"* It is very notable that while the arithmetic mean stays at 3 between the surveys, the standard deviation falls from 0.87 to 0.52. This is quite a significant change.

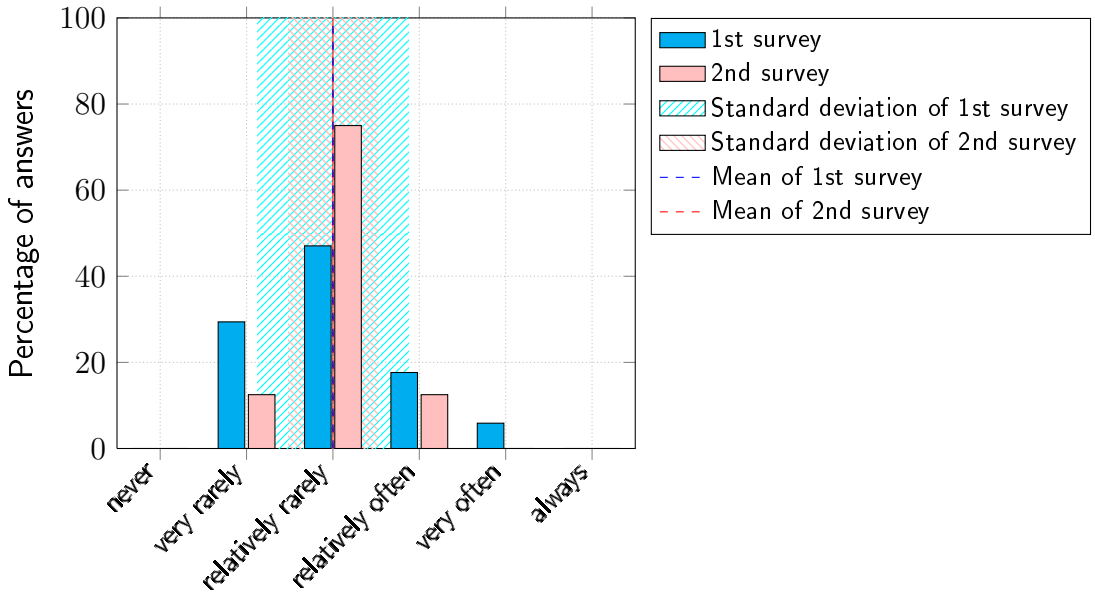

Figure 26: Difference in local defects caused by changes

While the average did not change, the difference between the developers grew noticeably smaller. This can be attributed to writing tests to cover the changed functionality and this in turn produced better quality code with less defects. It is worth noting, that it was not always possible to cover the changed functionality with tests, because of the architecture of the system and huge amount of legacy code.

The differences in question *"How often the changes you make create (including database changes) unexpected problems in somewhere completely unrelated part of software?"* between the first and the second survey are graphed in Figure 27. Both surveys had the answers clustered around *"relatively rarely"* with the second one being a slightly better. The difference is very small though. The standard distribution was smaller in the second survey: 0.75 versus 0.87. One explanation for these changes could be that the automated tests form a safety net that helps the developers to avoid introducing bugs. On the other hand the changes are relatively small and the amount of tests compared to the actual code so there most likely is no correlation here.

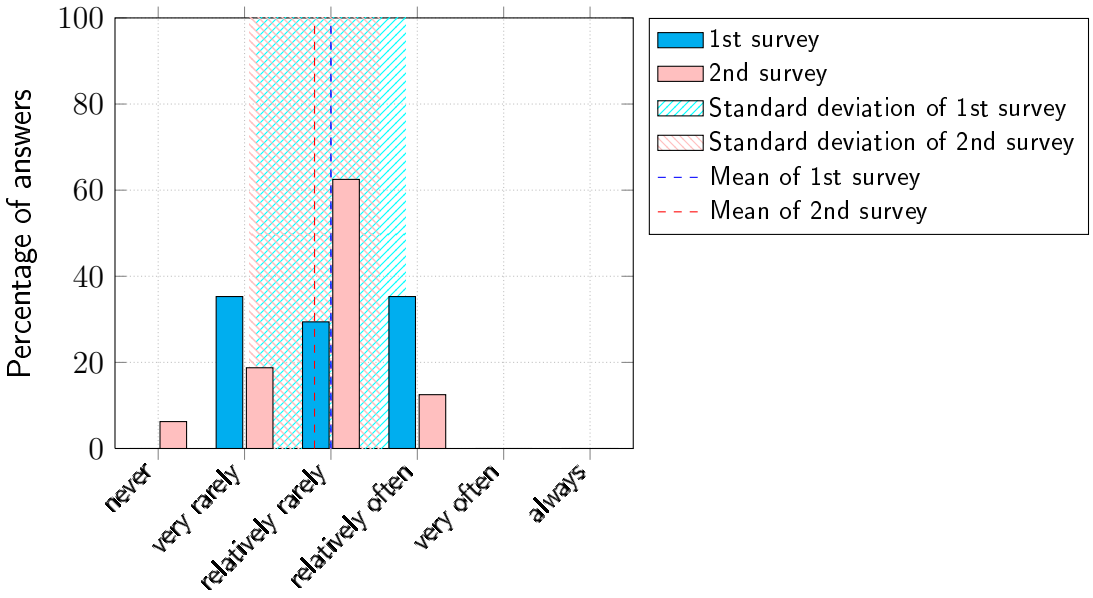

Figure 27: Difference in global defects caused by changes

One of the major findings is shown in Figure 28, which shows answers of both the first and the second survey to the question *"How often already fixed bug reappear?"*. While the arithmetic mean in both cases is around *"relatively rarely"* the second survey had much smaller standard deviation. The first survey had a standard deviation of 0.88 while the second one had only 0.43. This would indicate that the automated tests levelled the field between developers that are really well familiar with the system and those who have focused on a smaller area. This helps everybody since the likehood of the software breaking because of a change should be smaller than without the tests.

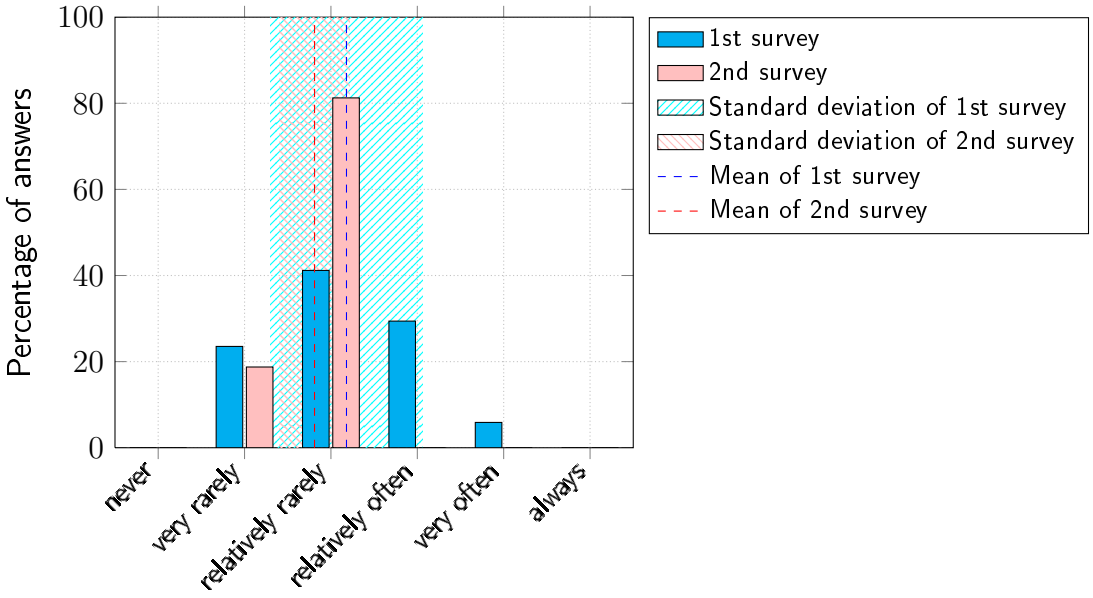

Figure 28: Difference in returning defects

When results from questions *"How easy it is for you to verify the changes related to rest of the system?"* and *"How often the changes you make create (including database changes) unexpected problems in somewhere completely unrelated part of software (so called house of cards effect)?"* are plotted on the same plot, one can easily see that there is a linear correlation between them. One can see from the graph that the feeling of something being difficult to verify is linked to that nagging feeling that there will be bugs left in the code. If this could be changed by some means the developers most likely would feel better when working on a complex software.

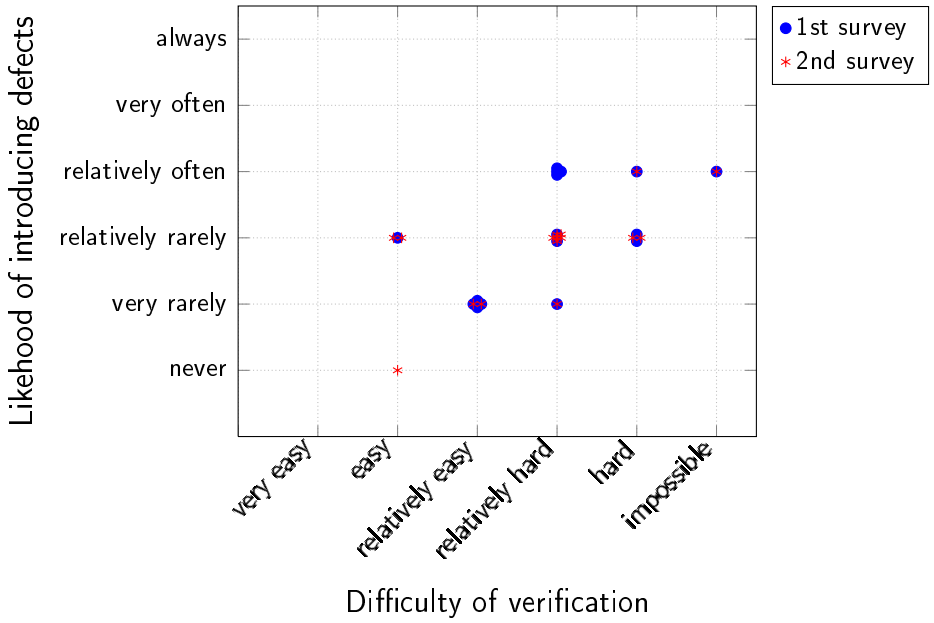

Figure 29: Correlation between the difficulty of verification and the likehood of introducing defects

A positive side in Figure 29 is that it shows how the developers' view has changed over time towards more positive aspect. A likehood of introducing bugs is less in the second survey than in the first one. Views towards difficulty of verification does not seem to have changed significally though.

15.5 Summary

It was notable that the same problems that Whittaker et al. (2012, 58) point out were noticed during the project: inertia, bad tests, no tests, testing is the problem of someone else. Even the smallest things seemed to take a long time to get moving, quality of the tests was not that good in the beginning and testing seemed

to receive only a half-hearted focus. But over the time as developers got started and understood the benefits, all these obstacles were crossed one by one.

The comparison between the two surveys show overall improvement and give a positive message regarding to automated testing that is performed by the developers. While the automation is time consuming and sometimes difficult it seems to help the developers to perform their work better and produce higher quality code.

The major improvement according to the surveys were in questions *"How hard is it for you to see how components work together?"*, *"How often the changes you make create (including database changes) unexpected problems in the functionality you changed?"* and *"How often already fixed bug reappear?"*. In the first case the arithmetic mean fell from *"relatively hard"* to *"relatively easy"* while in both the cases the standard ditribution was smaller.

16 Results

16.1 Comparison to Earlier Studies

In their study Williams, Kudrjavets and Nagappan concluded that in general, developers found unit testing worth their time and it helped them to find easy bugs before delivering the software to the testing team (Williams, Kudrjavets and Nagappan, 2009, 86). The results shown in Figure 31 from research done for the present thesis are similar compared to the results Williams et al. had, which are shown in Figure 30. The results of Williams et al. have a more positive view towards automated testing in general. Only in the statement *"Unit tests help me debug when I find a problem"* the research done in the present thesis showed that the developers value automated testing more than in the study by Williams et al..

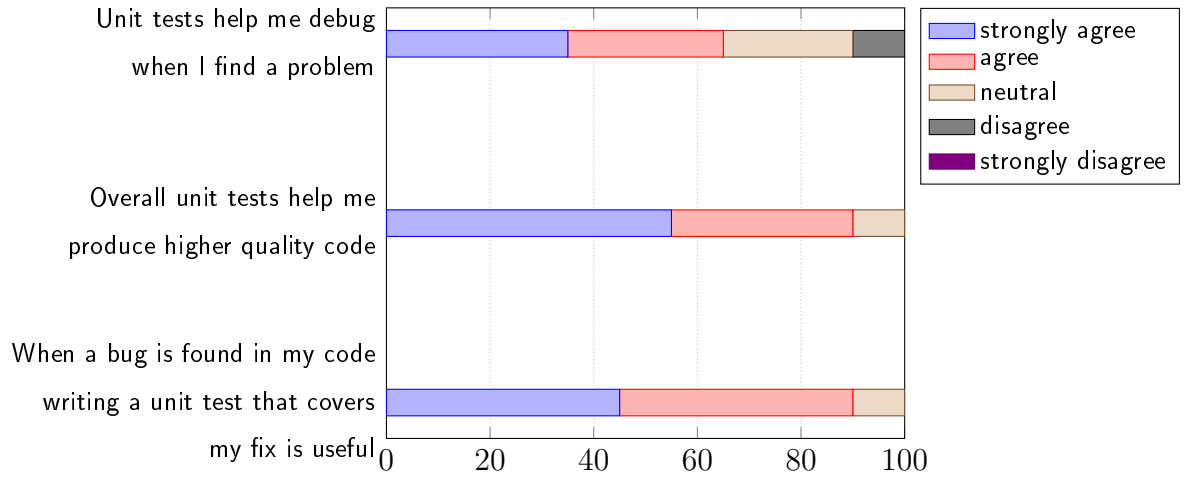

Figure 30: Developer perception (Williams et al., 2009, 87)

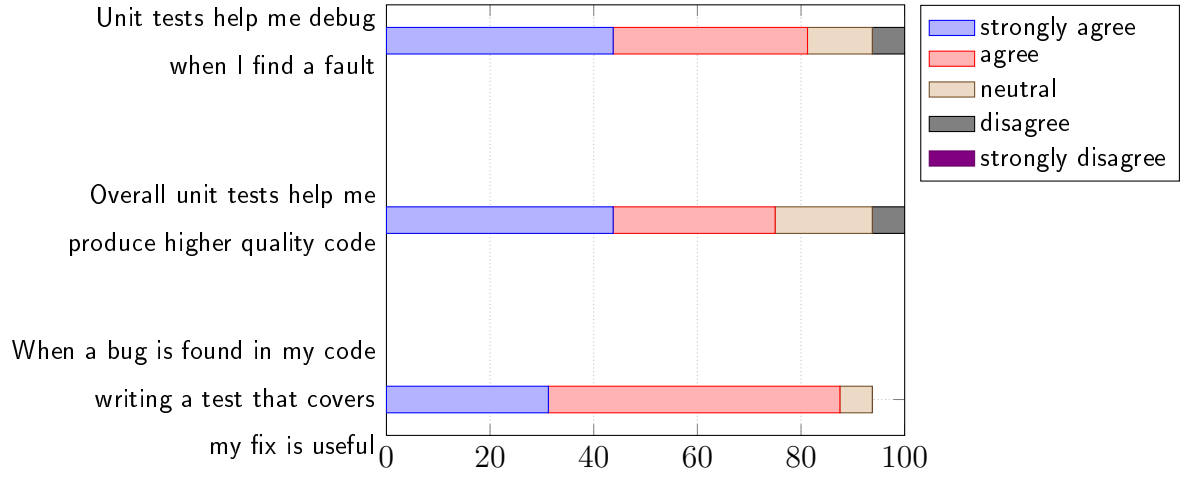

Figure 31: Developer perception at the commissioner

Williams et al. (2009, 86) state that the quality of the software was increased during the research; however, the development seemed to take longer. This is similar to the results in the present thesis. It depends on the case if the increased quality is worth the longer development time. Since the system that was under development by the commissioner has a very long life-cycle the tests are most likely worth the extra effort.

Writing automated tests might be a reason why the quality of code from different developers is more consistent (Erdogmus et al., 2005, 236). The results of the surveys would indicate similar effect, since in all questions the standard distribution was smaller in the second survey than in the first survey.

16.2 Limitations of the Surveys

The amount of participants in the research surveys were relatively small and the results might not be very conclusive because of that. Only roughly half of the people who were invited to participate to the survey actually answered, which might skew the results.

The research is based on the developers' subjective view and their opinions regarding automated testing and the quality of the system. While it can be used as an indicative of the quality in general it does not give the full picture. The system in question is very large and most likely the developers were working only on some parts of it. Their answers to the surveys might be affected by this.

17 Conclusions

17.1 Objectives of the Thesis

The first objective of the present thesis was to evaluate different ways of performing automated testing, map out some of the common pitfalls and offer possible solutions to them. This was achieved well. Chapter 5 outlines types of tests that were examined and taken into use in the company, while chapter 10 shows some of the common pitfalls and offers solutions to them.

Three different approaches for automated testing was taken into use: unit testing, integration testing and end to end testing. Since they focus on different aspects of the software system, they complement each other well. While unit tests help maintaining the internal quality of the software, integration and end to end tests help maintaining the external quality. As of writing the present thesis, the biggest return on investment is probably coming from the integration and end to end testing. They catch errors that have been introduced while the system is being developed. Benefits from the unit tests are realised in a longer time scale. While they ensure that single functions and algorithms work as intended, the unit tests also help to make the software loosely coupled and built from reusable components.

The second objective was to improve the quality of the software system. This was partly achieved as shown in chapter 15.4, where differences between the first and

the second survey are analysed. Especially the external quality of the software was improved as the difference in regression rate between the developers grew noticeably smaller. The fact that the developers feel they have harder time understanding local parts of the software system, i.e. the parts they are actively working with, is interesting. While this might be normal statistical variation it could also mean that the new technologies and ways of writing software are not yet familiar to them. Therefore it might be prudent to offer more training, although the real expertise is ultimately gained by working.

17.2 Future Use of the Results

During the time the research was ongoing there were multiple changes in the commisioner company and the organisation was focused on improving the quality of the software. Therefore all of the improvement cannot be attributed to the automated testing. This also highlights the difficulty of a research in the software industry: software projects are almost always one of the kind and are often initiated to create something that has not been built by the team before. Performing a research and analysing the results is challenging because arranging a control group is not always that simple. By performing multiple case studies in different companies a better understanding can be achieved; however, that requires a significant investment in time and money. By combining results from several different research it is possible to see if there are any major trends that are visible in most of them.

Analysis and comparison done in chapter 16.1 show that the developers in the host company have similar views towards unit testing as the participants in the study by Williams et al.. This might indicate a trend, but more similar studies would be needed to validate it. The results from the present thesis and the research by Williams et al. can be used in further studies of the same subject.

Analysis in chapter 15.4 can be used when a software company is evaluating advantages and disadvantages of automated testing. The analysis shows how difference in regression rate between different developers grew smaller and how changes caused fewer bugs in the changed functionality. Both are important factors for software quality. Again, if there is similar study done in a different company, the results from the present thesis can be used as a reference.

17.3 Further Subjects for Research

The domain of automated software testing is rather wide and sometimes very complicated. the present thesis could only touch some of the aspects related to it and further research and study would be needed in order to deepen the understanding regarding automated testing and its applications in the host company.

While there already are required tools for writing executable specifications in a form of acceptance level tests, their usage is mostly confined to developers. Because of this a really strong tool for bridging communication gap between developers and domain experts is not being fully utilised. It would be a good idea to continue work done by Koudelia and the author of the present thesis and try to get domain experts and developers to define the software system together, in form of executable specifications. Especially FitNesse is a promising tool for this.

A combined amount of tests in various tests harnesses at the time of finishing the present thesis was relatively low, only several hundreds. The time required to execute all of them was not yet a problem, even though most of them were run in a sequence on a single machine. As the amount of tests will grow, so will the time required to execute them. In the future it would be a good idea to do some research to identify and implement a solution that would allow executing tests in parallel and measure the impact on execution time.

In the host company, integration and end to end tests were executed against a known test environment that was always available. Generally it was not possible to have tear down methods to clean up a database after a test completes, resulting test database slowly accumulating a lot of data. The tests also had to be carefully written to check that certain configuration options stored in the database were in a specific state and change them if neccessary. This could be avoided by starting a new virtual environment for each test run and then discarding it after tests have been completed. Technology and tools for such a system are already readily available, but the space constraints prevented them to be addressed in the present thesis in suitable depth.

Narla and Salas write about hermetic servers that can be operated in a machine without network connectivity. Essentially the whole SUT is isolated, databases are

replaced with in-memory databases and externals systems are substituted with test versions. (Narla and Salas 2012.) Setting up a test system like this would be an interesting excercise and would probably result with faster execution of tests.

As more time passes and automated testing is treated more as a part of the software development cycle, the style of code is expected to somewhat change. It would be interesting to compare the code developed with the new methodology with older code and see how it has changed. One aspect of such a study could be collecting metrics automatically from the source code and calculating some key figures that measure complexity of the code and tightness of coupling between components and compare them with the old code. Because the old and the new code would be available along with the all the changes, the results could be plotted as a function of time.

17.4 In Closing

Large scale improvements that involve most of the development organisation are not easy nor fast to implement. In addition to the time and money they require an organisation that is very committed to improve and is willing to do the hard work required. Especially in the case of legacy software things are not always as easy as described in books.

In general, writing the thesis was a very interesting project and I learned a lot while working on it. In addition to the technical knowledge, I learned more about group dynamics and working as a member of a group of highly talented software professionals. Especially interesting it was to see how the automated software testing could be approached from different points of view and with different focal points. As mentioned in chapter 17.3 the domain of automated software testing is very broad. This in turn meant that the scope of the thesis had to be refined as the work progressed and many interesting subjects had to be left out.

Discovering that the developers viewed automated testing as a useful practice and that it actually improved the quality of the code was delightful. It is quite different to perform an experiment and measure the results than just read about them in a book.

While the present thesis is focused on automated testing the role of manual testing is equally important in software development. Regression testing for example is well suited for automation; however, e.g. usability and exploratory testing are something that computers are not capable of doing well. By combining human and automated effort with well planned and flexible ways, an organisation can most likely achieve better results than focusing only on one of them.

Bibliography

Digia - Annual Report 2012. (Accessed 1st of May 2013), URL `http://annualreport2012.digia.com/`.

FitNesse - One Minute Description. (Accessed 7th of October 2012), URL `http://fitnesse.org/FitNesse.UserGuide.OneMinuteDescription`.

SpecFlow - Pragmatic BDD for .Net. (Accessed 7th of October 2012), URL `http://www.specflow.org/specflownew/`.

SpiraTest Feature Tour. (Accessed 1st of May 2013), URL `http://inflectra.com/spiratest/Highlights.aspx`.

Cauldwell, P., 2008. *Code Leader: Using People, Tools, and Processes to Build Successful Software*. Wiley.

Denley, T., 2012. The Hamcrest Tutorial. (Accessed 7th of October 2012), URL `http://code.google.com/p/hamcrest/wiki/Tutorial`.

Erdogmus, H., Morisio, M. and Torchiano, M., 2005. On the effectiveness of the test-first approach to programming. *IEEE Transactions on Software Engineering*.

Feathers, M. C., 2011. *Working Effectively with Legacy Code*. Prentice Hall.

Fewster, M. and Graham, D., 1999. *Software Test Automation: Effective Use of Test Execution Tools*. Addison-Wesley.

Fowler, M. and Parsons, R., 2011. *Domain-Specific Languages*. Addison-Wesley.

Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D., 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Freeman, S. and Pryce, N., 2010. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley.

Graham, D., van Veenendaal, E., Evans, I. and Black, R., 2006. *Foundations of Software Testing: ISTQB Certification*. Cengage Learning Business Press.

Grubb, P. and Takang, A., 2003. *Software Maintenance*. World Scientific Publishing Co.

Hass, A. M. J., 2008. *Guide to Advanced Software Testing*. Artech House.

Hay, G., 2010. NHamcrest. (Accessed 8th of October 2012), URL `https://github.com/grahamrhay/NHamcrest/wiki`.

Holcombe, M., 2008. *Running an Agile Software Development Project*. Wiley.

Hutcheson, M. L., 2003. *Software Testing Fundamentals : Methods and Metrics*. Wiley.

Kaner, C., 1996. Software negligence and testing coverage. In *Software Testing, Analysis & Review Conference*.

Karten, N., 2009. *Changing How You Manage and Communicate Change*. IT Governance.

Koskela, J., 2012. *Test Automation and Robot Framework - Installation, testing and maintenance*. Bachelor's thesis, JAMK University of Applied Sciences.

Koudelia, N., 2012. *Acceptance Test-Driven Development*. Master's thesis, University of Jyväskylä.

Latornell, D., 2011. nosy 1.1.2. (Accessed 6th of April 2013), URL `https://pypi.python.org/pypi/nosy`.

Liskov, B., 1987. Data abstraction and hierarchy. In *OOPSLA '87 Addendum to the Proceedings*.

Loveland, S., Shannon, M. and Miller, G., 2004. *Software Testing Techniques: Finding the Defects That Matter*. Charles River Media.

Marick, B., 1999. How to misuse code coverage. In *16th International Conference and Exposition on Testing Computer Software*.

Martin, R., 1996. The interface segregation principle. *C++ Report*.

Melymuka, V., 2012. *TeamCity 7 Continuous Integration Essentials*. Packt Publishing.

Meszaros, G., 2007. *xUnit Test Patterns - Refactoring Test Code*. Addison-Wesley.

Moreira, M. E., 2010. *Adapting Configuration Management for Agile Teams: Balancing Sustainability and Speed*. Wiley.

Myers, G. J., Sandler, C. and Badgett, T., 2004. *Art of Software Testing*. John Wiley & Sons, Inc., second edition.

Narla, C. and Salas, D., 2012. Hermetic Servers. (Accessed 5th of October 2012), URL http://googletesting.blogspot.fi/2012/10/hermetic-servers.html.

Olds, D., 2012. How one bad algorithm cost traders $440m - a look at the worst software testing day ever. (Accessed 30th of March 2013), URL http://www.theregister.co.uk/2012/08/03/bad_algorithm_lost_440_million_dollars/.

O'Regan, G., 2002. *Practical Approach to Software Quality*. Springer.

Osborne, L., Brummond, J., Hart, R., Zarean, M. M. and Conger, S., 2005. Clarus: Concept of operations. Technical report, Federal Highway Administration.

Perkins, B., 2011. *Working with NHibernate 3.0*. Wiley.

Pohjolainen, P., 2003. *Ohjelmiston testauksen automatisointi*. Master's thesis, University of Kuopio.

Seemann, M., 2012. *Dependency Injection in .NET*. Manning.

Stephenson, A. G., Mulville, D. D. R., Bauer, F. H., Dukeman, G. A., Norvig, D. P., LaPiana, L. S., Rutledge, D. P. J., Folta, D. and Sackheim, R., 1999. Mars Climate Orbiter Mishap Investigation Board Phase I Report. (Accessed 29th of March 2013), URL ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf.

Taha, W., 2008. Domain-specific languages. In *The 2008 International Conference on Computer Engineering & Systems*.

Thomas, D. and Hunt, A., 2002. Learning to love unit testing. *STQE Magazine*.

Whittaker, J., 2011. How Google Tests Software - Part Three. (Accessed 6th of October 2012), URL http://googletesting.blogspot.fi/2011/02/how-google-tests-software-part-three.html.

Whittaker, J., Arbon, J. and Carollo, J., 2012. *How Google Tests Software*. Pearson Education.

Williams, L., Kudrjavets, G. and Nagappan, N., 2009. On the effectiveness of unit test automation at microsoft. In *International Symposium on Software Reliability Engineering (ISSRE) 2009*.

Appendices

1 Survey

Table 3: Original survey in Finnish

Minkä tiimin jäsen olet						
Aikaisempi kokemus automaattisesta testauksesta						
Arviosi automaattisten testien käytöstä tiimissä tällä hetkellä						
Kuinka helppo sinun on varmistaa muutostesi toimivuus?	Erittäin helppoa	Helppoa	Melko helppoa	Melko vaikeaa	Vaikeaa	Mahdotonta
a) Suhteessa muutettuun toiminnallisuuteen						
b) Suhteessa muuhun järjestelmään						
Kuinka vaikea sinun on hahmottaa?	Erittäin helppoa	Helppoa	Melko helppoa	Melko vaikeaa	Vaikeaa	Mahdotonta
a) Luokkien ja funktioiden toimintaa						
b) Komponenttien toimintaa suhteessa toisiinsa						
	Ei koskaan	Hyvin harvoin	Melko harvoin	Melko usein	Hyvin usein	Aina
Kuinka usein jo korjatut bugit tuntuvat tulevan takaisin?						
Kuinka usein ohjelmistoon tekemäsi muutokset aiheuttavat yllättäviä ongelmia (mukaanlukien tietokantamuutokset)	Ei koskaan	Hyvin harvoin	Melko harvoin	Melko usein	Hyvin usein	Aina
a) Muuttamassasi toiminnallisuudessa						
b) Jossain aivan muualla (ns. korttitaloefekti)						
Mitkä ovat tärkeysjärjestyksessä suurimmat haasteet liittyen testaukseen yleensä, mitkä olisivat parhaat keinot puutteiden korjaamiseksi?						
Vapaa sana						

Table 4: Translated survey in English

Which team do you belong to						
Previous experience related to automatic testing						
How would you grade current usage of automatic testing in your team						
How easy it is for you to verify the changes?	Very easy	Easy	Relatively easy	Relatively hard	Hard	Impossible
a) Related to other parts of the functionality						
b) Related to rest of the system						
How hard is it for you to see?	Very easy	Easy	Relatively easy	Relatively hard	Hard	Impossible
a) How classes and methods work						
b) How components work together						
	Never	Very rarely	Relatively rarely	Relatively often	Very often	Always
How often already fixed bug reappear?						
How often the changes you make create (including database changes) unexpected problems	Never	Very rarely	Relatively rarely	Relatively often	Very often	Always
a) In the functionality you changed						
b) In somewhere completely unrelated part of software (so called house of cards effect)						
In order of importance, what are the greatest challenges related to testing in general and what would be the best course of action to fix them?						
Comments						

2 Collated Data of The First Survey

subject;team;verify_local;verify_global;understand_local;
understand_global;returning_bugs;bug_local;bug_global

1;1;3;5;3;4;3;2;4

2;2;2;4;2;4;3;5;3

3;1;2;3;2;3;3;3;2

4;1;2;4;3;4;2;3;2

5;2;3;3;3;4;4;2;2

6;3;3;4;3;5;4;3;4

7;1;2;2;4;4;2;3;3

8;2;2;4;3;4;2;2;4

9;1;3;4;3;4;4;4;4

10;2;3;4;3;4;4;4;3

11;2;5;6;4;4;5;4;4

12;2;2;3;3;1;3;3;2

13;1;4;5;3;4;3;2;3

14;2;3;4;2;3;4;3;4

15;1;3;3;3;3;3;3;2

16;3;3;5;2;5;3;3;3

17;2;3;3;3;3;2;2;2

3 Second Survey

Table 5: Original second survey in Finnish

Minkä tiimin jäsen olet						
Aikaisempi kokemus automaattisesta testauksesta						
Arviosi automaattisten testien käytöstä tiimissä tällä hetkellä						
Kuinka helppo sinun on varmistaa muutostesi toimivuus?	Erittäin helppoa	Helppoa	Melko helppoa	Melko vaikeaa	Vaikeaa	Mahdotonta
a) Suhteessa muutettuun toiminnallisuuteen						
b) Suhteessa muuhun järjestelmään						
Kuinka vaikea sinun on hahmottaa?	Erittäin helppoa	Helppoa	Melko helppoa	Melko vaikeaa	Vaikeaa	Mahdotonta
a) Luokkien ja funktioiden toimintaa						
b) Komponenttien toimintaa suhteessa toisiinsa						
	Ei koskaan	Hyvin harvoin	Melko harvoin	Melko usein	Hyvin usein	Aina
Kuinka usein jo korjatut bugit tuntuvat tulevan takaisin?						
Kuinka usein ohjelmistoon tekemäsi muutokset aiheuttavat yllättäviä ongelmia (mukaanlukien tietokantamuutokset)	Ei koskaan	Hyvin harvoin	Melko harvoin	Melko usein	Hyvin usein	Aina
a) Muuttamassasi toiminnallisuudessa						
b) Jossain aivan muualla (ns. korttitaloefekti)						
	Täysin samaa mieltä	Enimmäkseen samaa mieltä	En osaa sanoa	Enimmäkseen eri mieltä	Täysin eri mieltä	
Kun korjaan järjestelmästä löytynyttä virhettä, minusta on hyödyllistä kirjoittaa testejä korjaukseni varmistamiseksi						
Testit auttavat minua kirjoittamaan laadukkaampaa koodia						
Testit auttavat debuggauksessa kun löydän ongelman						
Mitkä ovat tärkeysjärjestyksessä suurimmat haasteet liittyen testaukseen yleensä, mitkä olisivat parhaat keinot puutteiden korjaamiseksi?						
Vapaa sana						

Table 6: Translated second survey in English

Which team do you belong to						
Previous experience related to automatic testing						
How would you grade current usage of automatic testing in your team						
How easy it is for you to verify the changes?	Very easy	Easy	Relatively easy	Relatively hard	Hard	Impossible
a) Related to other parts of the functionality						
b) Related to rest of the system						
How hard is it for you to see?	Very easy	Easy	Relatively easy	Relatively hard	Hard	Impossible
a) How classes and methods work						
b) How components work together						
	Never	Very rarely	Relatively rarely	Relatively often	Very often	Always
How often already fixed bug reappear?						
How often the changes you make create (including database changes) unexpected problems	Never	Very rarely	Relatively rarely	Relatively often	Very often	Always
a) In the functionality you changed						
b) In somewhere completely unrelated part of software (so called house of cards effect)						
	Strongly agree	Agree	Neutral	Disagree	Strongly disagree	
When fixing a problem in the system, I find it useful to write tests to verify my fix						
Tests help me to product higher quality code						
Tests are useful while debugging, when I find a fault						
In order of importance, what are the greatest challenges related to testing in general and what would be the best course of action to fix them?						
Comments						

4 Collated Data of The Second Survey

subject;team;verify_local;verify_global;understand_local;
understand_global;returning_bugs;bug_local;bug_global;
fixing;quality;debugging
1;3;2;2;2;2;3;3;1;2;1;1
2;1;2;4;2;3;3;3;3;2;3;2
3;2;3;5;4;2;3;3;4; ;2;4
4;2;3;3;3;3;3;4;2;2;1;1
5;1;5;6;4;5;3;4;4;1;1;1
6;1;3;5;3;4;2;2;3;2;1;2
7;1;2;2;2;3;2;3;3;1;2;1
8;1;4;2;4;4;3;3;3;3;3;3
9;1;3;4;3;4;3;3;3;1;1;1
10;1;3;4;3;3;3;3;2;2;3;3
11;2;2;4;4;4;3;3;3;2;4;2
12;1;2;3;2;2;3;3;2;2;2;2
13;1;2;4;3;3;3;3;3;2;2;2
14;2;3;4;2;2;2;3;3;1;1;1
15;1;2;5;3;4;3;3;3;2;2;1
16;1;3;4;2;4;3;2;3;1;1;2