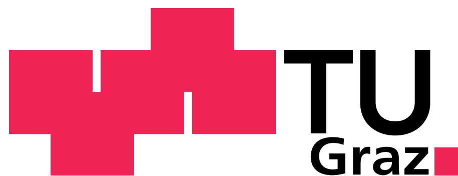
Stefan Stumpfl, BSc

Introduction of an Agile Development Management Technique Including a Test-Driven Development Paradigm on the iOS Platform.

Master's Thesis



Institute of Software Technology
Graz University of Technology
Inffeldgasse 16B/II , A - 8010 Graz
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Supervisor:
Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Graz, April 2014

EIDESSTATTLICHE ERKLÄRUNG

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

Graz, am ....

Name

STATUTORY DECLARATION

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

....

date Name

Abstract

Um den hohen Ansprüchen in der modernen Softwareentwicklung gerecht zu werden, ist die Einführung eines zeitgemäßen Managementprozess unausweichlich. In die Jahre gekommene sequentielle Entwicklungsmethodiken werden den ständig ändernden Anforderungen nicht mehr gerecht. Moderne und agile Methoden bieten eine vielversprechende Alternative und scheinen der einzig richtige Weg zum Erfolg zu sein. Hinzu kommen kontinuierlich ansteigende Qualitätsansprüche welche ein vollständig etabliertes Software-Testsystem erfordern.

Diese Arbeit befasst sich mit der Einführung einer agilen Softwareentwicklungsmethodik in einer Organisation. Weiters beschreibt sie die Etablierung eines vielversprechendem Testansatzes mithilfe von kontinuierlicher Integration.

Der erste Teil beschreibt theoretische Grundlagen zum Testprozess, wobei der Fokus auf testgetriebener Entwicklung mittels Unit-Tests liegt.
Im zweiten Teil wird die Einführung des agilen Entwicklungsprozesses auf Basis von Scrum und Kanban erläutert. Durch eine starke Einbeziehung des Kunden während des gesamten Prozesses wird versucht auf ständige Änderungen so schnell wie möglich zu reagieren. Kontinuierliche Integration der Software ermöglicht einen modernen automatisierten Testprozess. Besonderer Fokus liegt auf der testgetriebenen Entwicklung eines iOS Projektes.

Durch die Flexibilität des eingeführten Entwicklungsprozesses war es möglich auf ständig ändernde Anforderungen zu reagieren, wodurch ein sehr gutes Kundenverhältnis über den gesamten Projektablauf aufrecht erhalten werden konnte.
Der Ansatz einer kontinuierlichen Integration erlaubte die Anwendung von umfangreichen Testprozessen. Die Fehlerrate der Projekte konnte dadurch gering gehalten werden, wodurch ein hohes Vertrauen in die Software geschaffen werden konnte.
Bei der Umsetzung der testgetriebenen Entwicklung auf iOS gab es einige Schwierigkeiten, was zum größten Teil auf die fehlende Erfahrung und dem hohen Zeitdruck zurückzuführen war.

Schlüsselwörter: CI, Scrum, Kanban, testen, test, agil, unit-test, testgetrieben, software

Abstract

To meet today's challenging requirements in software development it is inevitable to establish a sophisticated management process. Due to constantly changing requirements it is no longer enough to simply follow an outdated sequential design approach. Modern agile management methods seem to be the winning way to satisfy today's demands.
Moreover continuously increasing quality standards call for fully matured software testing techniques to establish high reliability.

This thesis deals with the introduction of an agile development management method in an organization. Moreover it describes the process of the establishment of a promising test approach using continuous integration.

The first part describes the theoretical basics of software testing with particular focus on unit testing and test driven development.
The second part of this thesis deals with the introduction of an agile development technique based on a combination of Scrum and Kanban. The approach aims to a constant involvement of the customer, to ensure change requirements are recognized as soon as possible. Supported by the correct tools and a continuous integration process it ensures high quality software by automated testing. The work treats specifically the realization of a test-driven development approach on an iOS project.

By the flexibility of the introduced development technique, we were able to react on the constantly changing product requirements and thereby the relationship with the customer was excellent throughout the whole project cycle.
The established continuous integration introduced new opportunities regarding test execution, which helped us keep the error rate low and create confidence in our product.
Only the realization of the test-driven development approach caused problems, which was mainly because of missing experience and a constant pressure of time.

Keywords: CI, Scrum, Kanban, testing, agile, unit-test, test-driven, software

Contents

I Theoretical Background 1

1 About Software Testing 2

1.1 Introduction 2

1.2 Motivation 3

1.2.1 Why Should Software Be Tested? 3

1.3 The Testing Process 3

1.3.1 Who Should Test Software? 3

1.3.2 When Should Software Be Tested? 6

1.4 Different Testing Levels 7

2 Unit Testing 10

2.1 Introduction 10

2.1.1 Motivation 10

2.2 The Basics of Unit Testing 12

2.2.1 Definition 14

2.3 Core Techniques 14

2.3.1 Refactoring to Make Code More Testable 14

2.3.2 Indirect State and Interaction Testing 15

2.3.3 Stubs and Mocks 15

2.3.4 Isolation Frameworks 17

2.4 Managing and Organizing Test Code 18

2.4.1 Test Hierarchies and Organization 18

2.4.2 Naming Conventions 19

2.4.3 Ensuring Quality of Unit Tests 19

2.4.4 Unit Testing as Part of an Automated Build Process 19

2.5 Basic Pillars of Good Unit Tests 20

3 Test-Driven Development **21**
3.0.1 Advantages . . . 21
3.0.2 Disadvantages . . . 23
3.1 Techniques . . . 24
3.1.1 Test First . . . 24
3.1.2 Red, Green, Refactor . . . 25
3.1.3 Designing a Test-Driven Application . . . 26
3.1.4 You Aren't Gonna Need It . . . 27

4 Test Automation **29**
4.1 Introduction . . . 29
4.1.1 The Difference of Software Testing and Test Automation . . . 29
4.2 Motivation . . . 31
4.3 Disadvantages . . . 32
4.3.1 Problems When Introducing Test-Automation . . . 34

II Practical Part **36**

5 Introduction **37**
5.1 An Agile Development Method . . . 37
5.2 The Project . . . 38

6 Tools and Frameworks **39**
6.1 Jira . . . 39
6.1.1 Capabilities . . . 40
6.1.2 Advantages and Disadvantages . . . 42
6.2 GIT . . . 44
6.2.1 Successful Git Branching . . . 45
6.2.2 Advantages and Disadvantages . . . 46
6.3 Jenkins . . . 48
6.3.1 Architecture . . . 49
6.3.2 Setup and Configuration . . . 49
6.3.3 Encountered Problems and Pitfalls . . . 50
6.4 Unit Testing Framework . . . 51
6.4.1 OCUnit . . . 51
6.4.2 GHUnit . . . 52
6.4.3 OCMock . . . 52

7 Workflow and Process Automation **54**
- 7.1 Scrum 54
  - 7.1.1 Agile Workflow 54
  - 7.1.2 Different Roles 57
- 7.2 Kanban 57
- 7.3 Planning and Development Workflow 59
- 7.4 The Good Parts 60
- 7.5 Drawbacks and Possible Improvements 60

8 Test-Driven Development on iOS **62**
- 8.1 Getting Started 62
- 8.2 The Development Process 63
  - 8.2.1 Noticeable Lack of Experience 63
  - 8.2.2 Slipping Project Schedule 63
  - 8.2.3 Catching Up Writing Tests 64
  - 8.2.4 Conclusion 65
- 8.3 Ensure Testable Design 65
- 8.4 Why Did the Process Fail? 66

Part I

Theoretical Background

Chapter 1

About Software Testing

1.1 Introduction

Software testing generally is no new concept and has already existed with the introduction of the first computer programs. Back then, machine cycles where very expansive and software testing was mainly based on manually checking your code several times to make sure it will work afterwards on the machine, a process called ”desk checking” [1, p. xv]. Letting the machine do all of the work for you was not really imaginable. Why should anybody use the computer to check the programs, when we actually want to use its power to solve the actual problem. In the following years, when machines became faster, we completely discarded the idea of ”desk checking” our code. The practice of creating code increasingly became a trial and error experience. Just change the code and execute it until it does what it should do [1, p. xv].

Unfortunately we have lost some good discipline there. The desk checking practice per se has many advantages, it just takes too much time. Now that we have the computing power to let the machine do all this work for us, we should find a balance between then and now. Therefore we can write tests using the computer as resource and run them as often as the code needs to be executed [1, p. xvi].

1.2 Motivation

1.2.1 Why Should Software Be Tested?

Without any doubt the main goal for most software projects is to make profit. And in order to achieve this goal it is important that it provides enough value to the customer to convince them to buy the product.

The whole testing procedure which is involved in the software development needs to support the goal of making profit. If testing increases the costs, so that the software is not profitable anymore, it is not appropriate to do. But the tests can prove that the product provides the functionality the customer expects, which will improve the chance to convince him to buy the product. If you can't demonstrate the functionality, the customer may not buy it.

At this point it is a good moment to note, that the main purpose of software testing is not to find bugs, it is to prove that the software works. So we should think of it more like quality assurance instead of quality insertion. Finding errors is often associated with poor work at all, because it costs money to fix them. Money to fix problems which shouldn't be there in the first place. But as long as we are limited on time and resources, it is not possible to write software without any errors. So the best way to go is to find a balance between adding tests to control the development process and check the program to provide confidence that it works. The compromise should be based on reducing the risk to ship the product at an acceptable level. This means that the most critical components should be tested first then the next most critical. This should be continued until the remaining risk is not worth spending more time and money to fix it [2, p. 2].

1.3 The Testing Process

1.3.1 Who Should Test Software?

In the early days of software development projects were managed according to the "waterfall model" (see Figure 1.1). Even today, many projects are built based on these concept. The concept processes the different development stages step by step. Managers create product requirements and specifications which are then implemented by developers. At the end

the product is tested by an individual testing team [2, p. 2].

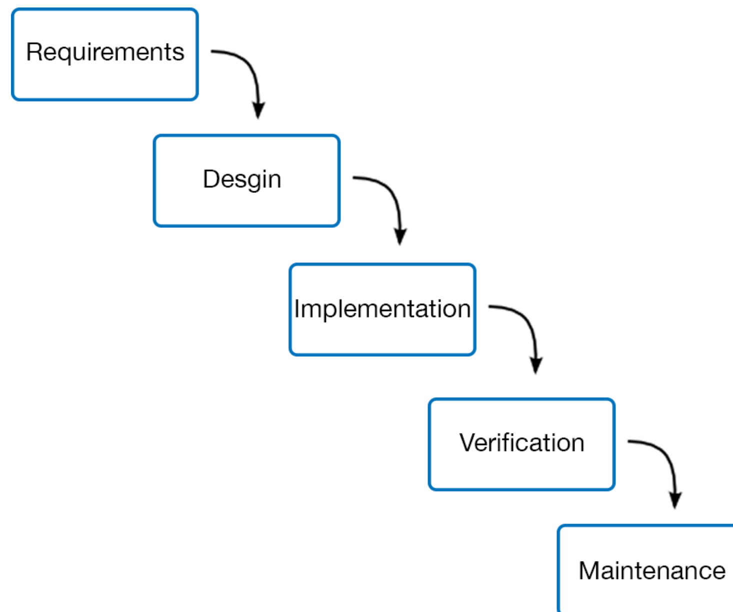


Figure 1.1: In the traditional waterfall approach each phase is performed individually. The method enforces a separation between testers and coders [2, p. 3].

This traditional approach separates testers and coders, which has some critical drawbacks but also benefits. One of those benefits is that individual testers are unaffected by any implementation details. Testers familiar with the code are often blind for many scenarios and won't find certain errors. Also it can help to uncover ambiguous specifications, which are often interpreted differently by unaffected testers. Misunderstandings concerning the requirements are a very common reason for failing tests.

The probably biggest problem for individual testers represents the fact, that the costs to fix an error will increase drastically with the time when it is found (as shown in Figure 1.2). Starting testing at the end of the project means, that every bug found has to be fixed then, which is the most expansive moment to fix bugs. This makes sense when considering that a bug found at the end needs to be reported by the tester first. Then the developer has to interpret the report correctly, reproduce the

error, locate it and find a way to solve it. Also, by the moment the error is found, much time has passed since the developer has created the code, which makes it even more difficult to locate and fix the problem. At the end the fix must be resubmitted and the application must be tested again [2, p. 3].

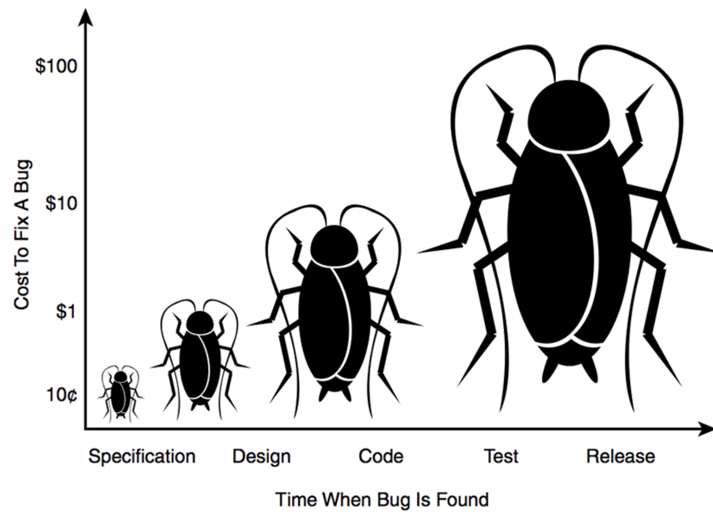


Figure 1.2: The costs to fix a bug will increase logarithmic. As in our example a bug found while developing may cost about 10$ while it will cost the tenfold or more in later stages [3, p. 18].

The costs of fixing bugs at the end of the development stage also depends on how early the bugs where injected into the software. In the worst case a misunderstanding of the requirements has the consequences, that a feature must be completely rewritten. This leads to waterfall practitioners following a very conservative approach, where requirements and specifications are completely overspecified. This attempt to avoid ambiguous specifications which can lead to misunderstandings is known as analysis paralysis and usually increases the project costs [2, p. 4].

External testers without any knowledge about the application internals follow a so called blackbox testing approach. Usually they perform system or integration tests which follow a strict plan. The testers take the specifications and create test schematics. The tests are often per-

formed manually as they have to be interpreted by the tester because of dependencies on external states [2, p. 4].

A very specific way of testing is done by beta testers. The involved people are no test professionals, they are mostly end users. The intend is to find problems with different system configurations between testers and customers. A very large number of users trying to find errors in special use cases which the team didn't thought of. Especially for small teams, which cannot afford individual testers, beta testers offer a chance to check the software.

One common problem with beta testing is, that the beta state means for the developers that the software is practically finished, and an end is in sight. Thereby the motivation to deal with beta feedback is not that high. But it is really important to deal with the problems found, since the whole time spent testing the software would have been wasted time if nobody is willing to fix the bugs.

The main part of testing is done by the developers themselves. Building and debugging the code while writing the software is a kind of testing. In addition, developers are inspecting code internals and will get rapid feedback in the form of compiler warnings. The advantages and disadvantages when developers are testing software is almost exactly the opposite of them for individual testers. They will find the errors much earlier and may know where the exact problem is located. There is still the possibility of misinterpreting the requirements which will not be discovered until a second tester is going to check this [2, p. 5].

1.3.2 When Should Software Be Tested?

As already mentioned in a previous topic, the earlier a software can be tested and bugs may be fixed, the cheaper it will be. It is good to know that software on which will further be built on is already tested and does work. This will result in less errors and a faster development process and thus is much better then testing all of the software only at the end.

However, software is traditionally tested mostly at the end of development. An explicit QA phase follows the development process. Then a beta release will be published where inexperienced testers can perform a last check on the software before a general release will be published. Most of the modern approaches already recognized this deficit and try to counteract this practice. Different approaches to continually test all

parts of the software during the development cycle become more and more popular. These define the main differences between agile projects and traditional managed projects [2, p. 6].

Agile projects are organized in short iterations called sprints. The project requirements will be reviewed after each sprint and obsolete stuff gets dropped. If any changes are necessary, they will be applied and the requirements and specifications will be updated. Then the current most important requirement gets designed, implemented and tested in the current sprint. A review at the end of the sprint will decide whether the implemented feature will be added to the product, or if it needs further changes. One important point is that the customer is included in all significant decisions, so there is no need for an overspecified requirements document. He will be asked how the application should work and has the opportunity to confirm or deny that it works as expected.

So in agile projects all aspects will be tested continuously. An extreme example for an agile development approach is called "extreme programming" where developers are working in pairs. One takes the control of the keyboard while the other one thinks about the implementation, how it could be improved and what kind of pitfalls may be involved. The whole process runs under a test driven approach [2, p. 6].

So the simple answer to the question when software should be tested is always. There will always be users who use the software in some unexpected way and they will discover bugs nobody was aware of. It is simply not possible to cover all circumstances, not with any reasonable time or budget. But it is possible to let the developer do all the testing for the basic stuff, leaving the QA team and beta testers free to try out the experimental use cases. So separate Testers will have the opportunity to find ways to break your application [2, p. 6].

1.4 Different Testing Levels

The main different testing types can be categorized under two basic approaches [22]:

Blackbox Testing This approach, also called functional testing is an technique which ignores the internal behavior and structure of the soft-

ware. The output for a specific input is validated according to the specifications. The approach is basically used for software validation.

Whitebox Testing Whitebox testing is also called structural or glass box testing. In contrast to Blackbox testing the approach focuses on the internal mechanism and structure of the program. This approach is often used for verification.

Considered in more detail there are the following widely used types of testing:

Unit Testing This technique describes the test of an individual program unit or a group of related units. Each unit should meet its expected behavior. It's mostly accomplished by the programmer himself and is one of the Whitebox testing approaches.

Integration Testing A group of components combined should produce an expected output. Integration testing also checks whether the interaction of hardware and software works as expected. It belongs to both, Blackbox and Whitebox testing.

Functional Testing The technique of functional testing is used to ensure that the specified functionality in the system requirements works as expected. It is an Blackbox testing approach.

System Testing For system tests the program under test is put in different environments, to check if the software still works. It is usually done with a full system implementation and falls under the class of Blackbox testing.

Stress Testing Checks how the system behaves under unfavorable conditions. Such tests are preformed beyond the limits of the specifications. Stress Testing is an Blackbox testing approach.

Performance Testing Used to test the speed and effectiveness of the program. It must be ensured that all results arrive within a specified time as it is defined in the performance requirements. The approach belongs to the Blackbox testing approaches.

Usability Testing Here the program is tested from the perspective of the user. It aims to check whether the GUI is user friendly and how deep the learning curve for the user is. Whether the product is pleasing to use in generally. Usability testing is also a Blackbox testing approach.

Acceptance Testing Acceptance testing is mostly done by the customer. It aims to verify that the delivered product meets the specified requirements. Does the program do what the customer wants? It is a BlackBox approach.

Regression Testing After the software has been modified or updated, regression tests should check if everything still works as expected. The technique is a Blackbox approach.

Beta Testing Beta testing is done by the end user after a beta version has been released. It aims to cover unexpected errors for special use cases nobody thought of. Since the user has no knowledge of any internals, it is also an Blackbox testing approach.

Chapter 2

Unit Testing

2.1 Introduction

Most programmers know that they should write tests, but hardly anyone really does it. Testing reduces bugs, provides basic documentation and improves the overall application design, but even though software testing is probably the most powerful tool to improve the quality of our software, it is really hard to motivate yourself writing tests.

2.1.1 Motivation

It can be really hard to find the motivation to write unit tests, and even harder to keep this up over the timespan of the whole project. But there are many good reasons to do so.

First of all, unit tests will change the way of how you are writing code. You don't have to think through the exact logic of the code you are going to write. You don't have to check the code afterwards multiple times and think about whether it will work or not. You just have to write the tests first and thereby define what your code should do. Then let the computer check your code. This can greatly speed up your development [4].

Tests will improve your overall design. Writing unit tests forces your code to be testable. It is really hard to write unit tests for tightly coupled

code, so you have to make your classes as loosely coupled as possible. Also you will rely less on patterns like global variables or singletons, as it can be hard to unit test such controversial design. In addition, writing tests makes you a user of your own API, so you start thinking how it should be used and how it could be improved [4].

Existing Tests will increase your confidence in the program. One of the biggest fears for developers is not knowing if all does still work after making some code changes. Unit Tests will take this fear from us. We do not hesitate anymore when changing and improving well tested code [4].

Unit tests will speed up the refactoring process or make it even possible at all. It can be really cumbersome to refactor code written long time ago, when there is no way to know if you broke something afterwards. Tests provide a great confidence that your changes don't break any expected functionality. Thereby you can refactor any time, which leads to even better code design. Also adding new features won't break any existing code without knowing [4].

What is the first thing you are looking for, when you have to work with a new API? For most people, the easiest way to learn is by examples. So most likely you are going to find code samples for the new API. This is exactly how your unit tests will serve as a documentation for your code. They will use your code the same way as anybody else is supposed to use it. You don't have to write boilerplate documentation nobody is going to read [4].

Unit tests will help you defend your code against other programmers. Imaging a bug which occurs only under very strict circumstances. In most cases it is very time consuming to find and fix such an issue. Now, any other programmer which doesn't even know the issue existed at all has to refactor some parts or add functionality. There are high odds he is going to reintroduce the fixed issue. You could have written an unit test which would have checked exactly this issue, so nobody would break your stuff again [4]. If anybody does, you would immediately know who broke it and can blame the responsible programmer instead of fixing your own stuff again and again.

In the end, unit testing makes development faster. Sure, on a class by class basis, testing slows you down. It takes some time to produce good tests, especially for beginners. But as time goes on and you are getting more confident in writing test, the velocity increases. At the end you will waste much less time on thinking about breaking code and can add features much faster as without tests.

It can also help to ignore internal implementation details on the first iteration of coding. Many people prefer a quick and dirty approach first. As long as we have tests it's no problem to come back later and improve or refactor your code [4].

And last, testing is fun! Good developers like challenges. It can be quite difficult to find well built solutions for automated tests. "Just like coding is an art, testing is an art" [4]. Do not make the common mistake to let the novice programmer do all the testing. Writing good tests needs experience, expert programmers are the best testers [4].

2.2 The Basics of Unit Testing

The test code invokes another piece of code, mostly a method or a function and checks if some assumptions about it are correct. It is important that the assumption concerns a single isolated code unit, thus the proven part must be completely independent from other logic. The unit test passes when the assumptions prove to be correct.

This describes more or less a typical way how unit tests are defined. It is strictly correct but lacks any advice what a good unit test looks like. To successfully apply unit testing to your project, it is not enough to only write unit tests. You have to write good unit tests or don't write them at all [1, p. 4].

To succeed in writing unit tests essential properties for a good unit tests have to be defined [1, p. 6]:

- It is easy to implement.
- Once written, it remains for future use.

- It is executable by anybody.
- It is automated and repeatable.
- It must run quickly.

Is it possible to write the unit test in a few minutes? If it takes more time to write the test, it is probably not a good unit test or not a unit test at all. Common reasons for such tests are internal and external dependencies, e.g. a database would be an external dependency. Such dependencies are often indicators for integration tests. Unit tests should focus on a single logical unit and dependencies should be stubbed or mocked, which should help writing them as fast as possible. The harder it is and the more time it takes to write tests, the less tests you will write [1, p. 9].

Is it still possible to run tests which I have written a long time ago? If we can't run them, there is no way to know if we broke any code we wrote back then. There could always break stuff when we are fixing small bugs at the end of the project. So we need to ensure within minutes, that everything is still working [1, p. 9].

Is it possible for anybody to run all existing tests? Everybody needs to execute all tests every time the code is edited. Otherwise there would be no way to know if anybody broke someone else's code. Without good unit tests there is always the fear to break someone else's code and thus most people avoid changing code at all [1, p. 10].

Are all tests automated and repeatable? It should be possible to run all tests with just one click. The more complicated it is to run them, the less often they will be executed. Nobody cares configuring settings to get the tests working. So they will probably be skipped at all [1, p. 10].

Is it possible to run all unit tests in a matter of minutes? The more time it takes to run them, the less often the unit tests will be executed. After code has been changed, feedback is required as soon as possible. The more time passes between code changes and the notice of broken code, the much harder it gets to find the responsible issue [1, p. 10].

If the answer to one of these questions will be no, there is a very high probability your unit tests are not that good. Or they are no unit tests at all [1, p. 7]. Unit tests which do not satisfy these properties will most of the time add more trouble than benefit and most teams will give up writing them [1, p. 5].

2.2.1 Definition

With the properties for a good unit tests specified we can form a new definition for those:

> *"A unit test is an automated piece of code that invokes the method or class being tested and then checks some assumptions about the logical behavior of that method or class. A unit test is almost always written using a unit-testing framework. It can be written easily and runs quickly. It's fully automated, trustworthy, readable, and maintainable."* [1, p. 10]

It's not always that easy to understand what kind of code should be tested. Let's describe the definition of a logical behavior more clearly:

> *"Logical code is any piece of code that has some sort of logic in it, small as it may be. It's logical code if it has one or more of the following: an IF statement, a loop, switch or case statements, calculations, or any other type of decision-making code."* [1, p. 11]

2.3 Core Techniques

2.3.1 Refactoring to Make Code More Testable

It is often the case that you have to open up your design in order to write proper unit tests [1, p. 77]. For example, tests need to have access to private properties and methods, sealed classes, non-virtual methods and it should be possible to replace dependencies via stubs [1, p. 78].

Object-oriented design demands the idea of encapsulation. So the basic approach for most of us is to hide everything that the user of our class does not need. And now our testing practice forces us to make all the internals visible again.

Many people think it is a bad idea to change the application design just to make it testable, but that's not the case [1, p. 77]. We need to look at our tests as another user of our API, then it is absolutely justified to adapt our design for improved testability. There are several ways to do so, which are called testable object-oriented design [1, p. 78].

2.3.2 Indirect State and Interaction Testing

The most common way to check, whether a method does what it should is to check its return value. But often it's not enough to do so or there is no return value at all. In such cases there are other ways to verify the expected functionality.

If there is no return value, we can check the result indirectly by observing the state of the class under test. This approach is called indirect state testing [1, p. 41].

In cases where this is not enough, we can look at the code under test in more detail and check whether it's correctly interacting with some other object. This so called interaction testing let us verify if some method has been called correctly. Usually interaction testing can be a little bit harder than state based testing [1, p. 83].

2.3.3 Stubs and Mocks

While small representative unit test examples in books usually get along with very simple techniques, that's not the case for unit tests in real world projects. The code under tests almost always relies on some other objects to work as expected. Typical external dependencies would be the local file system, a web service or dependence on memory or time [1, p. 49]. You have no direct control over them and the unit test would fail just because the web service is not available. The only way we can certainly tell whether our logic does work, is to force complete independence for our code under test while our unit tests are running.

So we have to find a way to control these dependencies. This is where we have to use stubs. Stubs are controllable replacements for any existing

dependencies [1, p. 50].

A mock object, although for many people quite the same as a stub object, has a complete different purpose. While we use stubs to avoid dependencies, we have to use mocks to verify our unit tests in special cases. There will be often the case where our code under test do not return any result or save any state we could check with our tests. In most of these cases the result of our logic will be a method call on another object. The process of testing such a behavior is called interaction testing [1, p. 82].

A mock object will serve as a replacement for the involved object. We have to tell our mock how the code under test is expected to interact with it and then verify this expectations. Our unit test will fail if the method was not invoked at all or was not called with the specified parameters.

As you should test only one responsibility in each unit test, it's important to have no more than one mock for each test. More mocks would mean you are testing more than one thing, which would indicate an overspecified unit test. You should identify what you actually want to test and use a mock object for this. For all the other fake objects you should use stubs and don't assert them [1, p. 86].

We can see that both, mocks and stubs are fake objects used to replace dependencies. But mocks can fail tests, while stubs can't, which leads to a complete different meaning for both. We can combine them as needed but we should never use multiple mocks or verify on stubs [1, p. 86].

There are several ways to replace our dependencies with stubs, for example we could adapt our design to use interfaces instead of concrete implementations and then replace the real object with the fake by dependency injection. This can be really time-consuming and forces you to use interfaces most of the time. Moreover it can get really difficult to write the appropriate mocks for each test as it can be hard to reuse them generally.

Instead there will be many frameworks, which will greatly facilitate the process of creating and injecting fake objects which will be described more detailed under the next topic.

2.3.4 Isolation Frameworks

Writing mocks and stubs manually can be really time consuming and comes with various challenges. To save time and avoid common mistakes it's recommended to use one of the various isolation frameworks out there. They can facilitate the process of creating mocks and stubs tremendous. Furthermore by using such frameworks the test complexity will come down and the errors can be avoided [1, p. 99].

Following there are listed a few typical framework capabilities:

- Dynamic fake object creation at runtime.
- Define stubs to return simulated values.
- Verify single or multiple mock calls.
- Mock parameter verification.
- Verify and triggering event-related activities.
- It must run quickly.

Beside all these advantages, there are also a few traps of which one should be aware of when using such frameworks [1, p. 136]:

- Easy creation of fake objects can lead to too much usage of them. This will worsen the readability of the tests.
- Verifying multiple things or use several mocks will lead to over-specified tests. These tend bo break on small code changes although the functionality would still be correct.
- Stubs should not be verified.
- Hard-coded values should be consulted for verification.

2.4 Managing and Organizing Test Code

As with production code, you have to manage and organize unit tests in the right way to ensure their quality. In order to reuse parts of the unit tests, you have to structure them correctly and build test hierarchies. Unmanaged tests will get unmaintainable and unreadable very quickly as the project extends [1, p. 141].

2.4.1 Test Hierarchies and Organization

One of the most important parts is to ensure the confidence into the tests of the development team. To get the most out of our unit tests, we have to run them as often as we can. The only way to get your developers to run them continuously is to make the testing process itself as fast and as easy as possible. When running all the tests takes several minutes, nobody is going to run them on every deployment. If they have to be configured in any way to be executable, they are not going to be carried out all the time. If a test is going to fail occasionally because of any dependencies or other circumstances, the team will lose confidence in them and is not going to rely on them anymore. Developers will get used to failing tests and they will probably ignore them in the future [1, p. 147].

Tests must be completely reliable. When a test passes, it means that the affected code does exactly what it is supposed to do. When a test fails, there must be any failure in the code under test [1, p. 146].

To ensure this behavior it is necessary to separate your test by their kind. Categorize them by fast running tests, usually these are unit tests, and tests that take more time to run, for example integration tests. This ensures that the unit tests can be run separately without any configuration needed and independent from any unwanted external influences. They will proceed quickly and thus can be run as often as possible. Integration tests will run less frequently, but they will run at least on nightly builds [1, p. 144].

To ensure that every developer in your team can execute all tests at any time, the tests must be managed under your source control. Usually all your tests will rely under a separate project and will be checked in with your production code every time [1, p. 148].

2.4.2 Naming Conventions

Beside the right location of your tests, it's important to give your tests meaningful names. The appropriate test code for each method should be easily findable. A common approach is to name your test project exactly as your main project prefixed with the name *tests*. Then create an own test class for each class with the same name suffixed with *tests*. It's required to be as specific as possible, so use the word majority of the word test to point to multiple tests in each class. The test method should contain the name of the tested method, the scenario for the test case and the expected outcome. For example *TestMethodName-Scenario-ExpectedBehavior*. If it's hard to name your tests because it does many things, it is probably time to split it into multiple test cases anyway [1, p. 150].

2.4.3 Ensuring Quality of Unit Tests

As with your production code, your test code is built up by object-oriented concepts. As such, it should follow the main principles of object-oriented programming. Code duplication should be avoided, so structure your tests to create reusable code. Use inheritance, extract utility classes and create factory methods. One common practice is the usage of setup and teardown methods [1, p. 151].

2.4.4 Unit Testing as Part of an Automated Build Process

Todays agile development processes requires to handle immediate requirement changes, which leads to the necessity of test automation. This usually involves a few common steps: The checkout of the latest version from the project repository. Then the code must be compiled and all tests executed. After it is ensured that nothing is broken, the code will be checked in to production. Usually a predefined build script will handle all this. If the build or any test fails, the relevant people will be notified [1, p. 142]. There is also much more stuff today's build server can handle, for example the representation of build and test metric and a complete history with backups and archives of intermediate builds [1, p. 143].

To start the automated process there are usually different ways to trigger the build. Nearly every build server supports continues integra-

tion, which means it will run this process continuously. The build could be triggered every time somebody checks in some source code or it could run every few minutes.

What kind of tests will be run on each trigger depends on the exact build type. Usually there are a few different kind of build types. This includes nightly builds for long running tests and system tests, release builds which add server deployment and archiving, or continuous integration builds which include all fast running tests [1, p. 144].

2.5 Basic Pillars of Good Unit Tests

Regardless of your test code management and organization, there are three basic principles which serve as indicators for good unit tests [1, p. 171].

Trustworthy Tests It's strongly required to ensure the confidence of your tests. Tests have to be free of errors and they have to test the right things.

Trustworthy tests let your developers know what is going on all the time. When a test fails, the code under test doesn't what it is supposed to do. When it passes, the tested part does work [1, p. 171].

Maintainable Tests Unmaintainable tests represent a high risk. As they won't get updated they will fail more often. As they won't get fixed, they won't get executed anymore at all [1, p. 171].

Ensure the maintainability of your tests is one of the most common problem, as it can get really hard to write maintainable tests with the project getting bigger and more complex over time [1, p. 181].

Readable Tests If you can't clearly understand your own tests months after you have written it, nobody else will either. Unreadable tests wont get fixed or updated. As you can't maintain unreadable tests, their trustworthy gets lost [1, p. 171].

Good naming of the tests and their assert message is inevitable for good readability. We should see our tests as stories we use to tell the following programmers about our code [1, p. 210].

Chapter 3

Test-Driven Development

In the early days of testing it was common practice to start writing tests not until the production code was finished. In recent times this has changed and tests are written more and more before the code under test is written. While the exact meaning of Test-Driven Development may vary, the term is mostly used for such a test-first approach [1, p. 16]. Figure 3.1 and Figure 3.2 show the differences between a traditional unit-testing approach and test-driven development.

3.0.1 Advantages

Test-Driven Development will significantly improve the overall code quality and object oriented design. Also the tests themselves will have a much better quality because of the way they are written. The number of bugs will decrease, which results in a increased confidence in the code. In addition, time to find bugs will be much shorter than before [1, p. 18].

Take the simple way Writing all the tests after the code would be exactly as bad as writing them all before any code has been written. People are much better in doing only one thing at a time, so try not switching context as long as you don't have to. Writing all code first and the tests afterwards means you have to deal with every problem twice. So write just one or even a few tests for a problem, then create the code to make them pass. Thus you can increase your efficiency and you can get the most out of your time [1, p. 13].

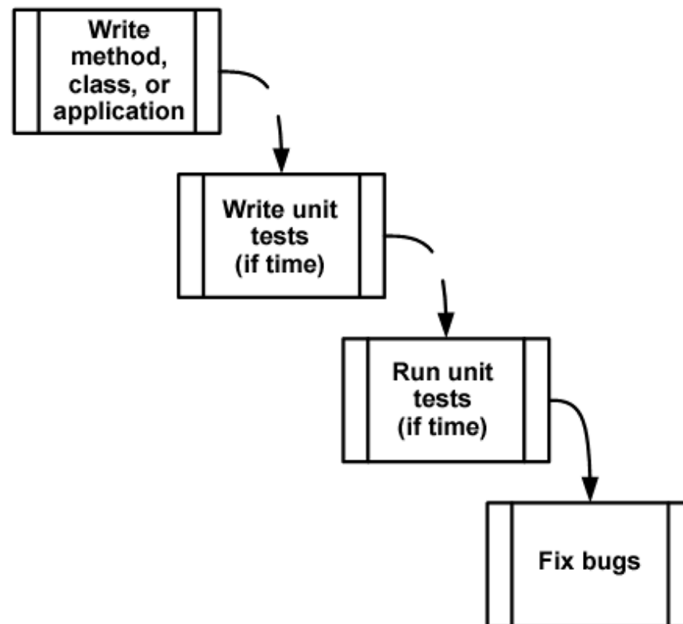


Figure 3.1: In a traditional approach most of the parts which involve testing are optional as represented by the dotted lines. If at all, they are written after some or most of the code is done [1, p. 17].

Increase your motivation When writing production code for your tests, you will get immediate feedback of your development cycle. Each test you can make pass will increase your encouragement. The process of TDD will split up large requirements in many small problems which you can solve in small steps. The permanent sense of achievement is a natural way to increase your motivation [5, p. 13].

Organize your thoughts You will have to think about the tests you are going to write. For each requirement you have to find the correct way to split it up in small problems and then write tests for these problems. You will only write production code if it is going to make a test pass, otherwise it is most likely not needed [5, p. 15].

Spend your time useful Much of the time needed for writing code includes time to test just written code. You need to get confidence that it does what it is supposed to be, because there are no tests which can tell you. Usually even more time gets lost finding small bugs and using

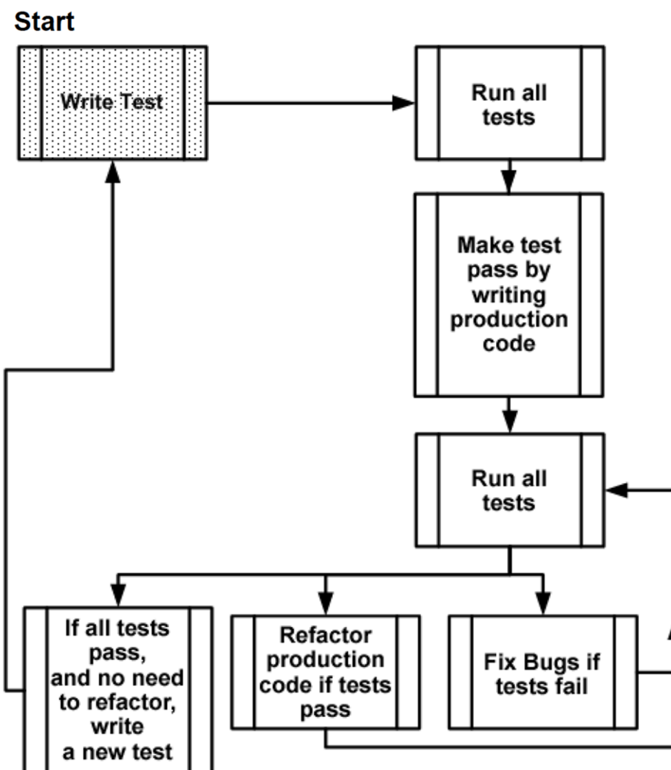


Figure 3.2: Test-driven development enforces small incremental steps. Unit-tests will be written first, then the code to make them pass followed by a refactoring process if necessary [1, p. 17].

the debugger stepping through your code.

After some experience with TDD, the total time spent writing tests and code isn't that much different to the time needed to just writing code. But the time lost by testing your code and using the debugger doesn't provide any value for the future. Unit tests on the other hand will stay there and will help you many times in the ongoing project cycle [5, p. 15].

3.0.2 Disadvantages

If Test-Driven Development is done wrong it can easily result in a slowed progress and thus it can delay the project schedule. It can waste developer's time and lower their overall motivation. Of course wrong tests

can also reduce the overall code quality.

One must be aware, that the advantages of a Test-Driven Development process come not without a price one has to pay. Mastering TDD requires a lot of experience, time to learn and time to implement your test environment [1, p. 18].

3.1 Techniques

The Test-Driven Development technique is based on three simple steps. First write a failing test, then write the code to make the test pass and in the last step refactor your code if this is necessary. The failing test is written as if code would be already working. It proves the absence of required functionality. A passing test means that the corresponding production code does exactly what it should do. The code should be written as simple as possible. Don't write more than you need to make the test pass. Refactoring is optionally. Sometimes it is enough to refactor after multiple passing tests have been written [1, p. 19].

The concept itself is not hard to understand nor hard to apply, but to succeed with it it's important to recall the unit testing patterns it is based on.

3.1.1 Test First

The unusual concept of the test first approach is to test something before it does actually exist. The test will define the acceptance criteria and as long as the test does not pass, the product needs to be further improved. As soon as all tests pass the project is finished [5, p. 13].

The problems usually do not describe user features. They are rather micro-features which form the functionality. [5, p. 14]

The best way to go is to not think about how to implement something. Instead think how you could write a test for the problem. This will lead to code which can be tested and will do what the requirement demands. As long as you think test-driven, the strict process when a test is written and how many tests are created before writing any code is not that important. Sometimes you will write a few tests and then make them pass. Another time you will write the code which should solve the problem first and then go back to write the tests to prove it [5, p. 14].

3.1.2 Red, Green, Refactor

Now that we know the basic concept of Test-Driven development, we need to think about how the tests should look like. It is important that the tests are derived from the specified requirements. Start with thinking about how you would like to use your implementation. What should the method call look like and which parameters does it need to solve the problem. Start implementing the method while leaving out any implementation logic. Then write the test which asserts the correct output.

The next step is to run the test. But why should we run the test knowing that we have not implemented any logic yet and the test will fail anyway? Sure the test will fail, but even failing tests provide value. They explicitly show that there is something the application should do but doesn't yet [5, p. 15].

Instead of just get going writing the implementation and thinking afterwards how it will be called and which parameters it does need, you have defined the call you would like to use first. This will greatly improve your overall application design. Your methods will be much easier to use for other developers. And you have shown that your project is not quite there, where it should be [5, p. 15].

At the early stages of your project this will be pretty obvious, but later on there will be cases where it is not that easy to know, what your application is already capable off. It may happen that you write a test with the intention to add a feature, not knowing that your application can already solve this problem. You will realize this not until you run the test and it magically passes [5, p. 14].

The process of writing tests although no code has been written, is often called the red stage. This is, because most of the modern development environments are displaying a red bar when any test is failing.

The red bar is only going to disappear when the failing tests have been fixed. There will appear a much friendlier green bar when all tests pass. This is the next step we aim to achieve. After defining the implementation interface in the first stage, we have to start implementing actual implementation. The curious part here is that it does not really matter how the code is written. You just have to make the test pass, even if this means that you just have to return the value as a constant for which the test assertion aims. As soon as the test is not failing anymore, you

have added the required functionality and reached the save green stage. Otherwise there would be a test to prove the contrary. Implementing anything more might just be a waste of effort [5, p. 16].

Now that you have implemented the required functionality it could be, that you do not really like your code. It seems that it is not as the object oriented world demands it to be. It can be really hard to tell what exactly is wrong, so it is used to be called that ”the code smells” [5]. This expression was invented by Kent Beck and refers to code, that may be okay, but there is definitely something about it that does not seem right. This is where you get the possibility to refactor your code. This should be simpler then ever, since you can't really break any existing functionality without knowing about. Your tests will provide confidence that your changes won't break anything. It is not required to refactor after each test. You can also do some refactoring after writing a few tests. You don't even have to refactor at all, if the code needs no refactoring [5, p. 17].

So there are three basic stages of Test-Driven Development. Writing failing tests is the red stage, make them pass is the green stage and the final stage is the refactoring process. Sure your application might not be finished after such an iteration, but it should be save to release an update because you have provided additional functionality without breaking anything else. Instead of implementing dozens of half baked features, we have defined a way to implement fully working and entirely tested micro features step by step. It has to be noted that there is still the requirement for other tests like integration and system tests before releasing any update of your software [5, p. 17].

3.1.3 Designing a Test-Driven Application

Although in the previous chapters we have learned how to implement features just by defining them through tests, it is not enough to just start writing the first test and incrementally adding all required functionality. Even Test-Driven development requires to think about your overall application design first. It is not that much different to physical engineering, where it would not be even thinkable to start adding components without thinking about the architecture first. You just can't

install your windows without putting in the walls first. But exactly this is what happens many times on software projects.

So before writing any test, start your Test-Driven project with at least an overall concept of your application. It does not need to be carried out to the smallest details. Just get an idea of all required features and how they fit together. Think about where common used code might be required and how your communication and interaction between the individual components can take place.

This knowledge will help you creating the correct application design while writing your tests. It is also very important for refactoring your code, to move your design in the correct direction [5, p. 18].

3.1.4 You Aren't Gonna Need It

One very important concept of Test-Driven development leads us to only writing code for tests we have written. And furthermore, only writing tests for features which are defined in the requirements explicitly. Thereby our project will contain no code we won't need, except for any requirement changes or other stuff becoming obsolete.

Usually developers tend to write more code then necessary. Why should I not implement nice stuff I'm absolutely capable off. It's often fun facing the challenge to write your code a little bit more generic or to extend your helper class with a few maybe useful methods. But a common problem with this is, that a lot of time gets lost adding this features. Your classes will degenerate to little frameworks and you have to invest much time just for the sake of completeness of this framework, time for implemented features you doesn't even need, at least not for this project [5, p. 19].

This is also often the case when implementing features while the exact requirements are still unclear. For example knowing that we are going to need a class which handles a web request to retrieve some information. We do not know anything in detail, so we could just start over writing a generic class, which should be able to handle all arising requirements. This is often called an inside out approach. As soon as we implement the exact details, we realize that most of the written stuff is not needed. But the code will be left there making it harder to read and maintain. This also comes with a bunch of tests we have written for this class. In the worst scenario the needless code may lead to an exploit by attackers. As

long as the code exists, there is also an eventuality that it may be used in the future. But this can easily lead to bugs which can be really hard to find since the code has been written a long time ago [5, p. 20].

Test-Driven development enforces an outside in approach. We will write only tests for things we will need for sure. It might be the case that you have to extend your code at some point to support features you would have already thought off. But since you got a complete test coverage it should be no problem to do so when the time has come. You can also write multiple tests and then write the code to implement it outright in a more generic way [5, p. 20].

Of course there are also cases where it is required to write generic classes. Many big software companies have to do so when building their frameworks. But they will invest much more time and effort to do so. Most of us write simple applications where no such requirement exists and it would be much better to spend our time on more important and really required things.

Chapter 4

Test Automation

4.1 Introduction

Automated software tests can reduce the testing effort significantly. Either the tests can be executed in much less time or the number of tests performed in a limited time will be increased. The required resources to execute them will decrease in any way. So test automation is a great way to save money on software development. Since it will require much effort to build up the automated tests, it will not always save money directly but you will get much better software quality more quickly.

Test automation builds up on the concept, that all test should run on the push of a button. So they can run overnight when computing resources are available. One of the main advantages is the repeatability. You can execute exactly the same test sequence with the same inputs over and over again. It would be really hard and time consuming to achieve this with manual testing. Even the smallest change in your production code can be tested with minimal effort. In addition it is great for your developers to get rid of boring repetitive test activities which are mostly error prone and discouraging [6, p. 3].

4.1.1 The Difference of Software Testing and Test Automation

To understand the concept of test automation it is important to see the difference between testing and test automation:

Software testing
There will be an unlimited number of possible test cases for any system and it is only feasible to create a very limited number of them. Therefore it is necessary to pick the right ones to build, the ones which will probably find the most errors. A good test can be described by four main attributes.

- **Effectiveness** indicates the probability of finding bugs.
- **Exemplary** tests will do more than one thing whereby the overall number of required tests will be reduced.
- How **Economical** is it to perform a test? How much resources are required?
- **Evolvability** is an indicator of the complexity of a software test. How hard is it to maintain the test, when software changes?

For example, a very examplary test will verify multiple outputs but may cost much more time to execute, analyze and debug. It's also very likely that such tests are hard to maintain. Therefore very examplary tests can drastically reduce their economy and evolvability [6, p. 4].

Test automation
Whether tests are performed manually or automated will make no difference in its effectiveness or how examplary it is. If your tests are not well suited for finding many bugs, they will neither do so when executed automatically. Automation will only affect the evolvability and economy of the tests. The costs of running all tests will decrease drastically but it will take much more effort to create and maintain them. It is important to have far reaching knowledge and experience in creating automated tests, otherwise the costs for creating and maintaining them can quickly exceed the costs of executing them all manually [6, p. 5].

4.2 Motivation

Test automation will help perform testing much more efficient than it could be done manually, but there are also many other advantages when using automated tests [6, p. 9].

One of the main advantages of automated tests is the possibility of regression testing. Tests can be performed repeatably for each new version of the software. After software changes, your tests will check the code for any introduced errors. This is especial useful in environments with a large number of programmers and permanent software updates. The extra effort should be minimal, since all the tests already exist and have been automated [6, p. 9].

The automated execution of tests will allow far more tests in less time. Thus they can be executed much more often which will lead to more confidence by your developers. It is often the case that automated tests do not lead to less time spent for testing. Instead much more tests will be executed more often [6, p. 9].

Test automation will enable the introduction of very large and difficult tests. For many tests, manually testing is not really possible or would at least require an extensive investment to do so. For example tests, where a large number of user data and their input is required to populate data required by the system. Manually creating hundreds or even thousands of user data is not really a possibility. An automated system can create these data within seconds, over and over again [6, p. 9].
Also, with manual tests you can only test a very limited number of attributes. For example a user interface action may trigger an event which has no immediate output. An automated system can easily check for such events [6, p. 10].

A further advantage directly affects the developers. It can be really boring to test code and simulate input data repeatedly. This can lead to unmotivated employees, which will probably raise the mistakes made by them. Automation will therefore increase accuracy and decrease errors. In addition, free resources can be used to create better test cases. Machines which would idle otherwise can be used to execute tests over night [6, p. 10].

Knowing that a large set of automated tests will run, leads to a greatly increased confidence by your developers. Newly released updates will be more robust, so any surprises after release are much less likely. This can drastically reduce the pressure on your team [6, p. 10].

With automated tests you can reach a consistency of test execution, which would not be possible otherwise. The tests will be repeated in the exact same order, with an identical input and at the same timeframe. This allows for the exact same execution on different hardware, platforms and operating systems and can lead to much more consistency for cross platform projects. By that, it will be much easier to enforce implementation standards across different platforms. The same feature will be implemented in the same way on each platform [6, p. 10].

And last, test automation can lead to a significantly reduced release time for your product. Since the test process will be integrated in the development cycle, the test procedure before release will be much shorter and the software will reach the market faster [6, p. 10].

4.3 Disadvantages

Besides the numerous benefits, there are also some disadvantages and common problems when trying to introduce test automation.

Automated tests cannot completely replace manual testing:
It would be whether possible nor is it recommendable to simply automate all tests. There are test cases which are just too difficult to automate. In such cases it can be much more economic to test things manually. Common examples for tests which should not be automated are [6, p. 22]:

- Tests which will not be repeated very often. It would be a waste of resources to automate tests, which will run only once every few months [6, p. 22].
- When software is changing very frequently, for example an user interface which is not yet final. It will be changed with each iteration and each automated test needs to be updated to. The effort to do this would be too high for its value [6, p. 22].

- For tests, where the verification of the result is an easy task done manually but would be very difficult to automate. Examples for this would be the esthetical appeal of an user interface, the correctness of images or an audio validation [6, p. 23].
- Mostly it is also not worth automating tests which require physical interaction. For example turning off and on a system or insert coins into a machine [6, p. 23].

At least at the beginning, all your testing should be done manually. Manually testing is much more effective in finding errors. You can then automate them over time. Begin with those, which are the most easiest to automate and which will run very often [6, p. 23]

Manual tests find more defects:
A test will reveal a defect the first time it is run. Before any test will be executed automatically, it has to be tested itself. This is done by executing it manually. Affected defects will be revealed just then. It is important to realize, that test automation is more for regression testing then for finding errors immediately [6, p. 23].

Automated testing requires a very high test quality:
Only the differences between expected and actual outcome will be revealed. It is important to have a great confidence in the correctness and completeness of all the tests which will run in the background [6, p. 23].

Automation doesn't improve effectiveness:
Automated tests are neither more effective nor more exemplary. They are just going to improve the overall efficients [6, p. 23].

Test automation can limit software development:
Usually automated tests are more fragile than manual tests. They tend to break by the smallest software changes. The effort in building and maintaining them can easily restrict the options for enhancing and changing the software. Changes which will have a strong influence on the tests should be eventually dropped [6, p. 24].

4.3.1 Problems When Introducing Test-Automation

There are some important points to consider, when introducing test automation. Since there are common problems depending on the project and the development team itself, it should be clearly thought of, if an automation will provide any value for your project [6, p. 11].

It is very common that the introduction of test automation digs to high and unrealistic expectations. It is the human nature to think, that new technology will solve all of our problems with minimal effort. Sellers of test automation software will withhold the given problems and disadvantages which will lead to unrealistic expectations. Expectations which cannot be reached will impact your motivation negatively on the long term [6, p. 11].

Test automation may not be a good idea if no experience is present and the testing practice itself is very poor. It is most likely better to invest more time in improving the test cases itself to make them more effective than improving the efficiency of poor tests [6, p. 11].

Just because the automated tests have not revealed any issue, this does not mean that there are no issues. Tests may be incomplete or incorrect at all [6, p. 11].

Tests require much effort in maintenance. When production code gets updated, many tests have to be changed to. If the effort maintaining your tests is greater then running them manually, automation will not provide any value [6, p. 11].

Test automation tools may be not free of errors themselves. They often require trained personal and support to know their strengths and weaknesses [6, p. 11].

There may be problems related to the software itself you are trying to test. If this was not written with testability in mind, it may be difficult to write tests. If you can't really test it manually, an automation tool is not going to help. It probably will make it just more difficult [6, p. 12].

It requires the right organization and extensive support by the pro-

ject management to integrate test automation successfully. The task is not trivial and requires much time, budget and trained personal. The team has to figure out a way what will work best for them. There should be one person with responsibility, which itself is enthusiastic about test automation. He needs to communicate his attitude to the team to motivate them. In addition, an appropriate infrastructure needs to be set up to make test automation possible [6, p. 12].

Part II

Practical Part

Chapter 5

Introduction

The main goal of this work was the introduction of an agile software development process for a new upcoming project. Once the technique has been established and well integrated into the company structure, the procedure should be adapted for other ongoing and upcoming projects as well.

The intended scope of the process should include agile development methods in a test-driven manner. Beside this, we had to set up tools to support this goal, including a proper project management tool, a bug tracking solution and a program to introduce Continuous Integration.

5.1 An Agile Development Method

We have decided to use an agile development approach based on Scrum. This widely used and very successful technique for agile development is based on short development cycles called sprints. Due to the fact that a requirement cannot be fully understood or defined, the approach follows the principle of a strongly involved costumer, who can track the project progress and guide its ongoing direction [7].

Although this flexible approach was very promising, it has been decided to not use it in its common definition. A very own process has been developed, with the intention to provide the best adaption for the team and the internal company structure. So the Scrum technique was combined with the so called Kanban approach. Kanban, literally signboard or billboard, is a system for a just in time production which has its

roots in the car industry. It uses a multi column board to schedule and assign the different tasks to the relevant people. The approach gives a great overview of how resources are currently spent and how the project progresses [8].

Our choice for a proper project management tool fell on Jira. Jira, a commercial, platform independent software product developed by Atlassian is one of the most widely used tools in modern software development. Beside project management it supports issue tracking and bug tracking. There are many different plugins, for example for Scrum and Kanban, which allowed us to setup our custom development environment. Furthermore, it can be integrated with all common source code management tools and it also supports native build server integration.

To establish Continuous Integration we decided to fall back on a widely used build server tool called Jenkins. Jenkins is a free cross-platform tool which was forked by Hudson, a widely known commercial build server. Since the development status for this tool seemed to be very active, and it supports iOS as well as Android and Web projects it seemed to fit perfectly for our needs.

5.2 The Project

The software project is about an online trading platform, where some kind of items can be collected and shared with other users. In order to use all offered features of this service, an user registration is required. The collectible items must be generatable with templates, which allows the customer to update and extend the item collections over time. The platform should be accessible via an easy to use web interface, as well as by different mobile clients. To offer the best possible user experience, a native implementation on each platform is a key requirement.

Chapter 6

Tools and Frameworks

As already mentioned we used Jira for project management, bug and issue tracking. Further, an important part was to connect Jira to our build server and source code management tool. Only in this way we could assure to gather all important information at one place.

So while looking for the right tools which would match our requirements, we have especially kept an eye on the ability to interact with each other. Thereby Jenkins became our choice for a continuous integration environment, while we decided to use GIT as our version control system. All of them promised to interact with each other quite good.

To support test-driven development for our iOS project we used a unit-testing framework called GHUnit and further a promising fake object creation tool called OCMock.

6.1 Jira

Jira is a commercial issue tracking software we also use for project management and bug tracking. The highly customizable program can be configured and accessed via an rich web interface. Its flexible plugin architecture allows for a selection out of a large number of third party plugins built by the Jira community. The available add ons helped us creating our agile board consisting out of an Scrum based approach combined with Kanban. Furthermore it is integratable with different source code management tools, so we could connect it to our GIT Server [9].

6.1.1 Capabilities

The following section lists features of Jira, which where an essential part for our decision to use this tool [9].

- The highly customizable Dashboard as can been seen in Figure 6.1 can be freely layouted to display activity stream, newest issues and tasks, latest bugs, current project progress and much more functionality. There is also the option to generate dashboard templates in order to provide predefined layouts for all users.
- It supports bug, issue, feature and task tracking. Although bug reporting can also be used from external testers and customers, we only used it internally since we only had a limited number of users. External bug reports where reported via emails, which where automatically populated into Jira using available plugins.
- A Jira agile extension enables the power of creating and estimating stories, building sprint backlogs, visualizing team activity and much more we required for our agile process realization.
- A substantial issue change history. Every changes made, will be logged and can help to understand past operations, which can be especially useful if somebody who was working on an issue has left the team.
- Jira has a paid extension tool to support documentation called Confluence. Confluence is a wiki we use to collaborate on writing and sharing content in our team.
- Widely available plugins, created by a large Jira community, enable a free customizable project management tool including the support to integrate external tools.
- Integration with GIT and Jenkins to support Continuous Integration. Test coverage and the latest commits can be directly displayed on the project overview.

Creating new issues, bugs or tasks can be done within a few steps. Although there are multiple fields and options when creation an issue, most of the time it's enough to fill out just a few of them, depending

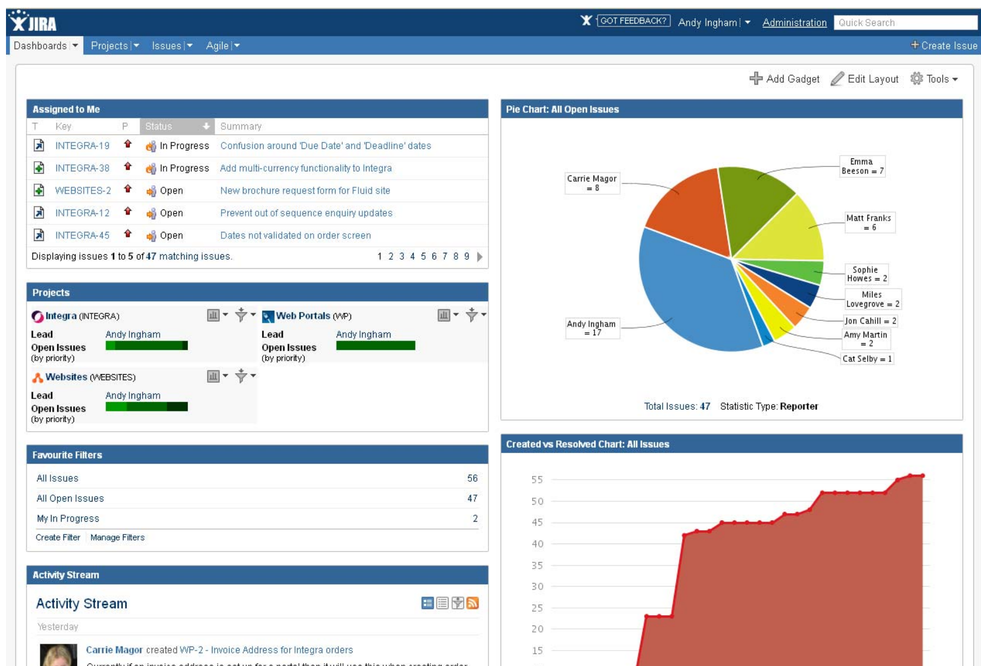


Figure 6.1: The exemplary Jira dashboard shows multiple projects, assigned issues and an activity stream. Additionally you can see charts regarding all different kind of issues and a comparison between created and resolved issues.

Figure 6.2: Apart from a few required fields like defining an issue type and a short description, there are a lot of additional fields e.g. priority, related component, affected version and many more.

on your own process flow. Required fields includes the affected project, the issue type and a short description for the issue. Figure 6.2 shows the creation of an issue including many optional input fields.

6.1.2 Advantages and Disadvantages

Jira offers a great opportunity to use one tool to fulfill nearly all requirements and wishes for project management and software development. A single place for everything, even if co-workers have to contribute from a completely different workplace in a foreign country. Also the Confluence extension is a great way to get all required documentation and information to one place everybody has access to.

Another great part is the email integration to report bugs, even

without any access rights. Especially end-users which won't have any knowledge about Jira, can report bugs per email, which will automatically appear in the internal bug tracking system.

Since the tool with all its possibilities can get really complicated, the ability to create templates for different purposes can greatly facilitate the usage for co-workers.

The support for a greatly customizable agile development interface allows the personal adaption as it's required for the internal company process cycle.

And last the possibility to integrate the tool with our SCM tool GIT and our build server Jenkins was one of the most important decision makers, since it extended the possibilities for our project managers to have all information at a single place.

Beside all the positive aspects mentioned, there are also some disadvantages when using this multi functional tool:

All the possibilities, customization options, plugins and extensions make the tool really complicated to learn and use. It can be hard for developers to get used to all the available functionality. In our experience most of them where satisfied with a very small part of the features, although there would be much more which could greatly facilitate their work.

Also the setup requires quite an expertise at this area. It's recommended to have a dedicated worker to explore and setup the tool as required. He should make use of templates to help co-workers get used to the tool. The tool is only really useful after correctly set up and customized as wanted. Much effort will be needed to find and install required plugins and integrate it with other tools. It is important to be aware of this facts when deciding for such a powerful tool. Especially for very small teams, this could be really hard to handle and not even necessary.

The structure of Jira can be quite confusing sometimes. The user-interface offers many options for the same use case, which makes it hard to remember the necessary procedures. Additionally there seem to be small notification issues, which can make the tool a little bit annoying at the beginning. For example the listed project lead will get an email notification for every single issue created, changed or removed. This will

result in spam with not much value. Furthermore, it's apparently not possible to get a notification if you got assign to an issue by somebody else.

6.2 GIT

The project is hosted on a dedicated in-house server with hourly backups to prevent any data-loss. We have also considered using some external hosting provider like GitHub but due to an restrictive contract with the customer, we where forced to ensure the project will be hosted internally.

For our source code management system we decided to use GIT. GIT is a distributed revision control and source code management system which has evolved as the most used and reliable source control system [10].

Although the previously used system was Subversion and there was not much experience with GIT at this point of time, we decided to use the newer alternative. Beside its popularity it was the extensive feature set and the possibility of an entirely new branching system which forced our final decision.

We have accomplished to integrate our GIT repositories into our Project Development Tool Jira as well as into our Build Server Jenkins.

This offers the ability to show each commit to its associated issue, bug or feature. Thereby the project lead has immediate insight who did the last update on an issue and what was it about.

The connection with our Build Server was crucial to reach our goal of a wide-ranging continuous integration. So we could configure Jenkins to react on new commits and trigger a scheduled test run.

Even though the vast set of GIT functionalities can impede the entry for beginners there, are plenty of free tools for every platform, which will simplify the usage at the beginning. Moreover a small set of features is already sufficient for an extensive branching approach as we have decided to use.

6.2.1 Successful Git Branching

To succeed with our Continuous Integration approach and to use the potential GIT entails in this regard, a successful branching strategy and release management was inevitable for us. So we've invested a lot of time and effort to completely understand the process of branching and to work on an approach which would be most convenient for our claims.

In the end we found an approach which was already successfully used by other development teams and also seemed to be perfectly suitable for our own goals.

Our approach is based on Vincent Driessen's famous branching model which is shown in Figure 6.3 and although we started with the intent to use it exactly as he described it, we did digress quite far with further project progress.

The repository setup is quite simple. There is one central repository referred to as origin, where every developer pulls new changes from and pushes the latest commits to. Actually GIT is a decentralized version control system, so technically there exists nothing like a central repository, but there is the option to create a bare repository which contains no working directory and that is exactly what we speak of [11].

With GIT it would be also possible for all developers to interact directly with each other and push or pull from their peers to. This could be especially useful for larger features but was not really used in our team, since the mobile development teams are quite modest and features were deliberately kept small [11].

Main branches: Main branches are characterized by an infinite lifetime and are referred to as master and develop. The master always reflects a production ready state of our product, while develop is the state with the latest delivered development changes. This would be also often referred to as integration branch, since it is fully functional all the time and automatic nightly builds are built from this.

After each new release the development branch gets merged into master and tagged with a proper release number [11].

Supporting branches: Besides the two main branches the approach includes a variety of parallel branches called feature-, release- and hotfix

branch.
The feature branches are used to develop new features for upcoming releases. They exist as long as the feature is in development and if successfully, it will be merged back into the develop branch, otherwise it will be discarded. Mostly there is a large variety of such branches at the same time but they reside usually only on the developer repository and not on the origin itself.
Release branches are used for last minute changes and bugfixes before a new release takes place. Only features targeted for the new release will be included. The branch will then be merged back into develop and master.
Hotfix branches are for situations when a critical bug in the production version needs to be fixed immediately. One person from the team will create a new hotfix branch out of the master and will prepare a quick hotfix, while the rest of the team can continue with their work completely unaffected. The branch will be merged afterwards back into the master and also the develop branch [11].

6.2.2 Advantages and Disadvantages

There are many discussions on advantages or disadvantages of GIT compared to other version control systems. But since you can find more than enough of these in selected literature or on the web, we will just focus on our decisive points.

First I would like to mention that the branching opportunity GIT allowed combined with a promising branching system, already caused major euphoria in our team, even before the project development has started.

- If everyone follows the rules and won't push any untested code to the development branch, there is never again the possibility that anybody broke something by which all other developers would get hold up. It won't be the case anymore that anybody checks out some code in the morning and has to wait until some teammates show up, since nothing is working anymore.

- Also on planned releases the rest of the team can continue working on their features since the branching system grants them complete

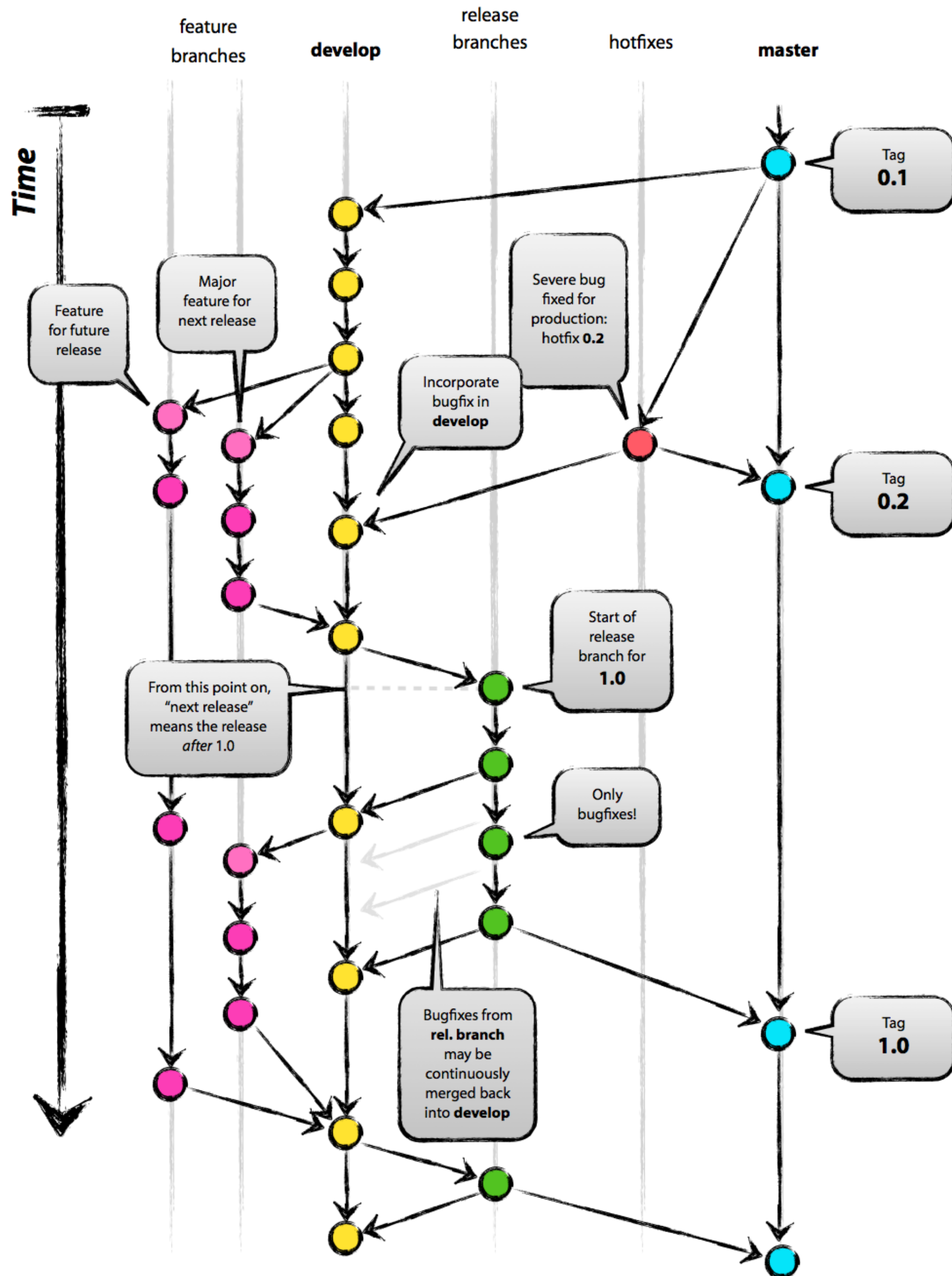


Figure 6.3: The model shows multiple branches used to ensure a successful branching and proper release management. Beside master and develop, which exist permanently, there are also a few branches with a very limited lifetime [11].

independence.

- Hotfixes can be applied to the production state without major circumstances, and they will also be added to the current development state.

- The branching allows developers doing experimental stuff which can be immediately discarded afterwards.

- And the last but probably most important point is the support for Continuous Integration which gets completely new opportunities due to the applied branching.

But there are also a few cons on the list which should be mentioned to:

First the branching seems to be a bit excessive for very small teams. Although parts of the model should be used even if somebody is working on a project alone, the system can cause much overhead and the excessive branching can lead to many merge conflicts if not done right.

To avoid such conflicts generally the features should be kept quite small and it should be pulled and pushed to develop as often as possible but at least once a day.

Even though the branching model worked very well, it was sometimes just too much overhead for our very small mobile development teams. In addition it can be very hard to run a very excessive branching on iOS development since the framework makes it very hard to merge changes if not done continually.

6.3 Jenkins

Due to its wide distribution we decided to use Jenkins as our Continuous Integration tool. Jenkins is open source and can be used to build and test software projects continuously. It supports a various number of different SCM tools and can execute builds by many different practices. To start a

build a few approaches exists, for example it can be triggered by a commit of a version control system or triggering can be scheduled repetitive by a predefined timespan.

The application scope can be extended by a large set of plugins. They can add additional build options, add functionality or integrate further version control systems.

Reports of successful or failed builds can be directly reported to Jira, where they will be displayed in the activity stream [12].

6.3.1 Architecture

Since our project includes multiple applications on different platforms, it requires a slightly more verbose Jenkins architecture to support integration for all of them.

Mainly there are three dedicated Jenkins server. One master running on a Linux machine and two slaves, one running Mac OS X for iOS application builds and the other is a Windows based machine executing builds for dedicated Windows applications. The master has set up different jobs to start a various number of builds for different platforms. It is constantly checking the GIT repositories to trigger builds on new commits. The affected clients are cloning the relevant projects from their respective repositories and then execute them via terminal. On a successful build, both unit tests and user interface tests will be executed. The build and test results are then reported back to Jira.

6.3.2 Setup and Configuration

The complex architecture requires much effort to set up all Jenkins machines and configure their desired jobs. Besides setting up the interaction between master and slaves, it is also necessary to find and install the proper plugins to ensure the interaction with GIT and Jira. The master-slave connection will be established over a ssh communication.

For each job it has to be defined on which Jenkins note it will run, what repository and which branches it should check out and build and to which branches a successful build must be merged back. Additionally the desired build triggers have to be defined. To execute the project and run the tests, for each job a special build script needs to be created. The

responsibility of these scripts is to start the build via console, execute all configured tests and write back the test results in a proper format.

Further it's possible to define post build actions which should evaluate the test results and notify the appropriate clients. In our case we had also a Jira build reporter which automatically created the corresponding issues if a build or test has failed and assigned it to the build breaker.

In the end, when all the Jenkins clients where successfully up and running, the master has constantly checked the different development branches for changes. As soon as any changes where registered, the corresponding client has checked out the newest changes, created a built and executed all tests. If no errors occurred it merged the changes back to the development branch. In case anything went wrong it forwarded the information to Jira, which on the other hand sent out emails to the affected people.

6.3.3 Encountered Problems and Pitfalls

As with Jira, also the Jenkins setup and configuration requires enormous time and effort and there should be definitely an individual worker who is responsible for setting up and configuring all required modules. Even if all is up and running, it still requires permanent maintenance and fixes for occurring issues. Plugins are often only barely reliable and it requires further effort to fix problems or you have to wait for updates which fixes them.

While the build server was running pretty well for the web and android based projects, it was really embarrassing to successfully setup all the required stuff for the iOS project. First it was troublesome to setup the master-slave communication on the Mac OS X machine. The requirement for administrator rights requires any user authentication which was not even possible when adding the relevant ssh keys.

A similar problem appeared when executing XCode builds with the console. Since Jenkins was running on the machine as an individual user, there were no rights to create any builds.

After these problems have been wiped out, there were already more troubles waiting. Since there is no official support to build iOS projects from console and executing their unit tests, much time and effort was

spent in creating the appropriate build scripts to do so. Even the task of generating the test results in a proper format was not that straight forward as it should have been.

The entire process was influenced by researching how to bypass any occurring problems. Most of the time, if a solution has been found, it was already outdated and did not work as expected, because XCode or any other part of the framework got updated which destroyed the approach. In the end after enough time and effort has been spent to bring the integration up and running there is usually already a new update from Apple waiting which is probably going to destroy much of the used effort.

6.4 Unit Testing Framework

To sustain the test-driven development process on iOS we decided to use any tools and frameworks which can facilitate our approach. There are a few different unit-testing frameworks for iOS including a native supported one by Apple called OCUnit. Beside some frameworks based on a behavior-driven development approach, there seemed to be only one widely used alternative called GHUnit.

To avoid writing required mock and stub objects manually, we decided to use OCMock since it had all usual required functionalities.

6.4.1 OCUnit

OCUnit is the default framework provided by Apple. Since it directly ships with XCode, it is probably the most easy choice to get started. It is based on SenTestingKit, which is simply a revision of Kent Becks unit-testing framework for smalltalk [13]. XCode versions prior to 4.0 had some poor limitations regarding to SenTestingKit, for example the inability to set any breakpoints in test cases which would be essential to debug failing test cases. Since then Apple provided incremental OCUnit updates to make up for most of the crucial drawbacks which made it a much better choice to use.

Drawbacks: There is no appealing user interface for running the unit tests and the output is limited to log results in the console. The probably biggest problem is the inability to run individual tests or a predefined set of test cases.

6.4.2 GHUnit

GHUnit is an open-source framework for writing unit tests on Mac OS X or iOS. It can be used as a standalone version or may be combined with other testing frameworks such as OCUnit [14]. Since it's not implemented into XCode it's not exactly that easy to setup as Apples alternative, although the installation process is equal to other external frameworks as it can be installed with Cocoapods. Since OCUnit had many drawbacks in its first iterations, GHUnit evolved to a first choice alternative with a solid feature set and a committed community. There are also instructions how to integrate GHUnit into Jenkins but at the end it was not that easy to do so.

Strengths: Its strengths are clearly in the execution of test cases and in the sophisticated ability to display test results. You can either execute your tests by command line or using a tool including a welcome user interface. In addition to the option of running all tests at once, they can be executed individually or in a selected set of tests. The output is colored, filterable and navigable. There is even an option to display rich test metrics [14].

6.4.3 OCMock

OCMock is an open-source Objective-C mock object framework. It facilitates the mock and stub object creation on the fly. Expectations can be defined with a syntax equal to standard method calls [15]. As with GHUnit the setup is quite easy since it can be installed using Cocoapods. The framework supports stub objects to return defined values for specific method calls, dynamic mocks to verify interactions and partial mocks in order to overwrite only specific methods of an existing object.

Useful features [15]:

- Supports mocks and stubs for class methods.
- Arguments on mock objects can be constraint or even validated using Objective-C blocks.

- Nice mocks allow ignoring unexpected method calls with the possibility to disallow specific invocations.
- Partial mocks will forward not stubbed calls to the original object, other calls will be handled by the mock object.
- Method swizzling to exchange specific methods of an object at runtime.

There are no real drawbacks using this tool. The fact that on iOS projects views and their controller are usually tightly coupled, can make writing unit tests really difficult. Many framework interactions in each method require multiple stubs to simulate the correct values. Writing all the stubs and mocks manually wouldn't really be an option. Overall OCMock provides great value and is a pleasure to use.

Chapter 7

Workflow and Process Automation

It was important for us to not just mirror or follow the process of a widely used Scrum definition. We were focused to find our very own approach of agile software development, which would match our internal company structure and which would fit best our needs. Therefore our approaches for Scrum and Kanban will differ from traditional methods.

7.1 Scrum

Scrum is a technique for management of agile software development. It is an iterative process which focuses on a strong involvement of the costumer. By that, the project risk can be reduced as the interaction with the customer will result in flexible changes of the requirements all the time. Furthermore it encourages teams for self organization with a very own approach of project management [7].

7.1.1 Agile Workflow

Our project was mainly split up into three different sub projects, one for each platform. For each a Jira project was created which leads to individual scrum procedures for each of the projects. At the start a main pool of user stories for each project was created by a very small team including the costumer. The team consisted of the product owner, the

scrum master, an individual project lead for each project and of course the customer.

After this initial setup more user stories were worked out and existing stories were refined. The involved people where constantly changing, depending for which project stories were created. Only the product owner was an exception, since he was involved in the story creation process all the time.

User stories are reflected in our Jira setup as epic stories, while more detailed tasks on which developer could work on are called stories (see Figure 7.1). The elaboration of concrete stories was done by the affected developer team including their project lead. They got estimated in story points and assigned to an individual developer. Prior to an upcoming sprint all stories in the backlog were evaluated and the next most important got selected for the next iteration. In the course of this the stories are flagged with "selected for development" whereby they appeared in our initial column of our Kanban board in Jira.

Our sprints took up from one to four weeks, depending on the requirements for the next iteration. After each sprint we organized meetings involving all project teams to get an insight on what was going on for everybody. Within each project team including the project lead, weekly meetings made sure, that everybody was aware of the current project progress and what kind of problems occurred. Only in very critical project phases we held daily stand-up meetings to discuss who is working on which part, when he is planning to finish and what problems are currently critical. For these kind of meetings the number of attendees was very small what helped to save time and get things done.

Unfortunately we often undervalued meetings at the end of each sprint. We didn't discuss enough about the last iteration and what could be improved. Maybe this would have helped us to further refine the process for the upcoming sprints but as with most of all real world projects there was a constantly pressure to prepare the product for release in time.

jirapark Dashboards Projects Issues Agile Tempo Structure Create issue Quick Search

2stic.ios User-Stories

MyIOS

Search +Next Issue +Sub-Issue Edit Unresolved

Key	TP	Summary	Fix Version/s	Assignee	Issue T	Resolut	Component/s
TSTICIOS-170		<= Release 1.1	Release 1.2	Stefan Kohl (Inac		Fixed	Core
TSTICIOS-409		<= Release 2.0.1	Release 2.0.1	Wolfgang Lierzer		Fixed	
TSTICIOS-121		Der User soll nach der Registrierung und Wahl der ersten Collection ei	Release 2.0.1	Stefan Stumpfl		Unresolv	Collection Viev
TSTICIOS-200		Es wird eine konkretes Design der Hilfe benötigt. - 1) Unter "[Nam	Release 2.0.1	Daniel Mitteregge		Fixed	Collection Viev
TSTICIOS-212		Die "App Tour" soll nach der Registrierung bzw. dem ersten Login	Release 2.0.1	Georg Bachmann		Fixed	Collection Viev
TSTICIOS-341		Die "App Tour" soll über die Settings auch später abrufbar sein.	Release 2.0.1	Georg Bachmann		Fixed	Collection Viev
TSTICIOS-127		In den Settings soll der User eine Übersicht der editierbaren Themen s	Release 2.0.1	Markus Weinländ		Unresolv	Profil
TSTICIOS-204		Die notwendigen API's müssen vom Server zur Verfügung gestellt	Release 2.0.1	Michael Geier		Fixed	Profil
TSTICIOS-312		Integration der Edit-Profile API Calls.	Release 2.0.1	Markus Weinländ		Fixed	Profil
TSTICIOS-128		In den Aktivitäten erhält der User eine Übersicht der allgemeinen (Colle	Release 2.0.1	Georg Bachmann		Done	Aktivitäten
TSTICIOS-124		Über ein linkes Swipe-Menü soll ein User die wichtigsten Bereich der A	Release 2.0	Georg Bachmann		Fixed	Core
TSTICIOS-114		Als User will ich, wenn ich die App öffne und noch nicht eingeloggt bin,	Release 2.0.1	Markus Weinländ		Fixed	Welcome Scre
TSTICIOS-115		Als User will ich, am Welcome-Screen die Wahlmöglichkeit zw. Registri	Release 2.0	Markus Weinländ		Fixed	Welcome Scre
TSTICIOS-117		Als User will ich mich mit E-Mail-Adresse, Benutzername & Passwort re	Release 2.0	Markus Weinländ		Fixed	Login / Registri
TSTICIOS-120		Der User soll nach Registrierung seine erste Collection wählen müsser	Release 2.0	Stefan Stumpfl		Fixed	Collection Viev
TSTICIOS-213		Es existiert ein Screen in welchem ich eine neue Collection starten und	Release 2.0	Stefan Stumpfl		Fixed	Collection Viev

Showing 485 issues Info

Bug tracking and project tracking for software development powered by Atlassian JIRA (v6.1.4#6159-sha1:44eaede) · About JIRA · Report a problem

Figure 7.1: The nested structure of epic-stories and sub-stories permits a good overview. There is also the option to directly create, delete and move stories from there. Furthermore the issue state can be directly changed which will help selecting issues for the upcoming sprint.

7.1.2 Different Roles

Since the project was split up into three separate parts we settled with the following different roles:

Product Owner: Our product owner was in continuous talk with the costumer. He was involved in every crucial decision and helped creating all of the user stories. He acted across all projects and was regularly attending sprint meetings to retain a complete project overview.

Scrum Master: We had one scrum master with decent knowledge of Scrum and Kanban. He was strongly involved in setting up the required project management and agile tools. In addition he got significant expertise in software architecture and was mainly responsible for the setup of the architectural environment.

Project Lead: In contrary to the usual Scrum approach we also got an individual project lead for each of the three sub-projects. He was attending in all of his project related meetings and guided the project in the right direction. He was responsible for the main task of creating stories and assigning them to the appropriate developers. In addition he served as QA authority. Beside this, he was very helpful in solving problems within his team and took over the task to motivate all team members.

Development Team: Each subproject had a team of developers which were mostly professionals on their field. The web team was with four to six members the biggest group, while in the smaller mobile teams usually two or three people were working.

7.2 Kanban

Kanban defines an evolutionary change management which means that it tries to improve the overall process by small evolutionary steps. This requires the introduction of a visualized workflow including the tasks to be done and occurring problems. The visualization can be realized with a simple whiteboard which contains sticky notes representing unfinished tasks. In our case a proper Jira plugin mirrored these functionality. Cre-

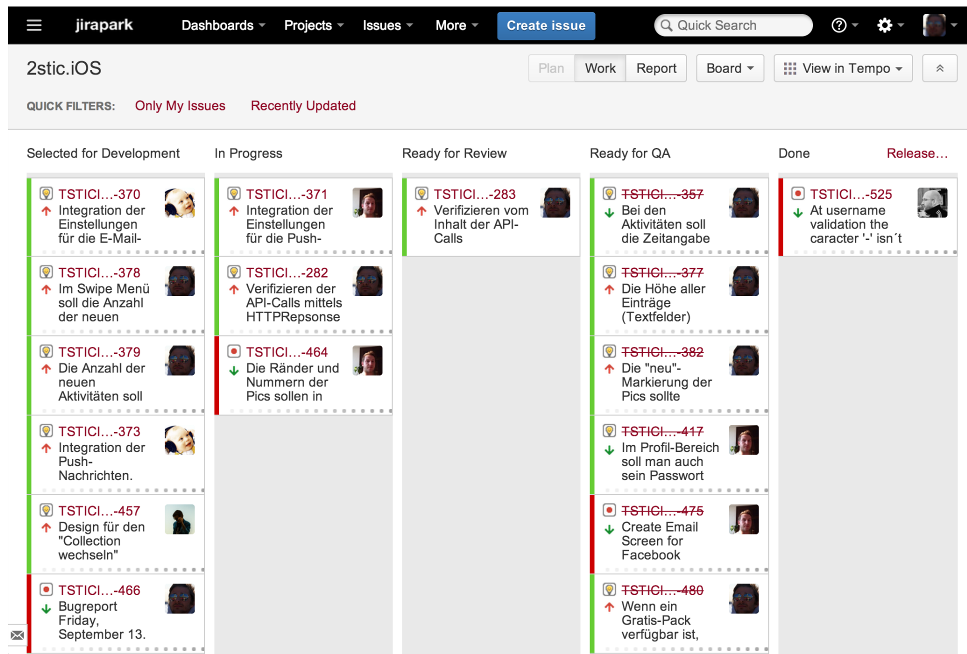


Figure 7.2: The board displays Jira issues selected for the current sprint. The different columns give a great overview of the current progress. Many problems in a single column reveal a possible bottleneck which needs to be addressed.

ated issues and bugs have been added to this Kanban board. The visualization in form of a public board is crucial to establish more transparency of how work is distributed and were bottlenecks may occur.

A further step is to limit the work in progress. This means limiting the tasks each team member is working on simultaneously. Thereby multitasking can be reduced and each individual task will be completed much faster. Also this enforces an immediate treatment of problems instead of working around of them constantly. This is realized by assigning an allowed number of tasks for each column.

At the end this should result in a constant flow of tickets in your Kanban board which live up to the core concept of these approach: No bottlenecks and no accumulation of tickets at any phase [8] .

7.3 Planning and Development Workflow

Product planning phase:
First user stories are derived from the product specifications. This results in Jira epic stories and is done by an internal team including the customer.

Concrete stories which can be assigned to individual developers are created by the desired development team including their project lead. All the stories will appear in the Jira backlog and stories selected for the next sprint will be tagged with "Selected for development" by which they will be listed in the Kanban board. An issue should be already assigned to a concrete developer by then. Possible estimation properties like story points can be added when creating the stories but this is not a requirement.

Development phase:
The responsible developer selects an issue from the Kanban board according to its priorities. For that purpose he moves the issue from the column "Selected for development" to "Under development". In the next step he has to create an individual feature branch, which will be named according to the issue's unique ID in Jira.

Before starting writing tests or any implementation, the concrete concept will be discussed with team colleagues.

Then the developer starts writing tests and implementation. After these are finished and local tests run successfully, he will push the feature branch to a central repository.

Our build server Jenkins will recognize these changes, clone the new branch and execute the build and all its tests.

Only if all these tests succeed, the implementation has to be reviewed by some other team member. For this the developer has to move the issue in the Kanban board from "Under development" to "Under review" and assign a concrete reviewer. Also a merge request for the reviewer will be created.

If the reviewer has nothing to add or change he will move the issue from "Under review" to "Ready for QA". Then he has to merge the new feature in the development branch where everybody can check it out.

7.4 The Good Parts

Most of the planned process did work out quite well so I will focus on a few points which went particularly good and provided great value to the project:

- By the weekly meetings including the costumer it was able to guide the project in the right direction all the time. There were not any bad surprises for the customer which resulted in a very good relationship and a pleasant atmosphere over the whole project cycle.
- Daily stand-up meetings in critical phases provided much value since they helped to identify possible problems and redistribute resources. Also it was important to keep the number of participants small to ensure fast and uncomplicated meetings.
- Reviews before and after implementation can indeed take much time and be a little annoying but they will provide much value. It can help prevent misunderstandings, improve the overall application design and bugs often will be found much earlier. Furthermore it will result in much cleaner code since everybody knows that somebody else is going to review the code which forces him to conform to the coding standards and keep the code clean. And last it can provide much value for less experienced developers since they can be guided by experts.

7.5 Drawbacks and Possible Improvements

There were also some drawbacks and things which could be improved, although most of the negative parts are related to the fact that it was not always easy to motivate yourself to comply to all those process steps all the time.

- After some time the process of creating user stories has changed to allow individual developers split up or create new stories. Only the developers knew what stories a concrete feature required. Further, due to our branching model it was important to keep the stories small to avoid substantial merge conflicts.

- The work of creating concrete implementation stories was often a little bit annoying, so it became more and more the responsibility of the developer to create them. Since they got enough other things to do they were not pleased to do so.
- Unfortunately we did not use the opportunity to discuss things which could be improved in the meetings after each sprint. Maybe this could have helped to improve some stuff during the project cycle.
- Not matter how important the reviews are, and how much value they will provide in the long term, it is often very annoying for the developer to get interrupted for the sake of doing code reviews. We should have thought about things to motivate the developer for doing code reviews.
- Each column in our Kanban board had a very limited number of allowed tasks. But this was only in theory. Since there was no real limit the phase of reviewing and doing QA spilled over most of the time. Often nobody was willing to do the boring tasks until they were forced to.
- Since Jira is not always easy and straight forward to use, most of the developers felt it's to much overhead to do all the required steps in Jira. They got already enough other stuff to do and tools to interact with. Maybe a cleaner and simpler user interface could improve this. Or even a few short trainings would help to get the developers comfortable with the project management tool.

Chapter 8

Test-Driven Development on iOS

8.1 Getting Started

After all of our used frameworks for testing were up and running the next step was to get a little bit more of knowledge regarding the whole iOS framework and also about test-driven development. Since I had no experience and knowledge with iOS development I started reading some books and created a few example applications. Though, my experience in the area of test-driven development was slightly more, again I was no more than a beginner, so I started reading more books about it, especially in the context of mobile development.

The intention to realize the project in a manner of pair programming was quickly abandoned, especially by our project lead because we had little time and not enough resources in terms of developers.

Most of the time we were two developers creating the iOS specific part of the project. After we thought trough the overall application design and built up some basic concept we were ready to start implementing our first iOS application in a test-driven manner.

8.2 The Development Process

Since the framework makes it sometimes really hard to work together on the same parts of the application, especially on view and controller logic, we split our work up into two separate areas. While one was focusing on creating some models and business logic, the other started implementing view and controller logic, at the beginning mostly by using some dummy data.

8.2.1 Noticeable Lack of Experience

At the beginning the progress was very sluggish, mostly due to missing experience. Especially for the part of view and controller implementation a test-first approach seemed to be harder then thought. The closer you get to the view layer the more interactions with the framework are necessary, which makes it really difficult to write unit tests. For the part of the business logic it was a little bit less annoying to write tests first.

How are you going to write tests for functionality you have no idea how to implement it? The fact I didn't write any real iOS application before complicated the approach tremendous. We had to invest much more time and effort to create the application as we had estimated before.

After our schedule was already slipping we tried switching more to a code first approach and writing test immediately afterwards, at least for our view and controller logic. Fortunately our OCMock framework helped us immense while creating fake objects for all the unavoidable framework interactions. But even thereby the number of required mocks made our unit tests more and more cluttered and unreadable. Of course this was not beneficial for their maintainability.

8.2.2 Slipping Project Schedule

Since our progress was much less then calculated we were by far not there, where we should have been at this time. The first beta release of the application with a very limited scope of functionality moved closer. So we had to remove the amount of unit testing to get back into our schedule.

For some time we completely stopped the process of test-driven development and we just created immediately required functionality.

Of course, as soon as you stop completely unit testing your application logic, most of their advantages get lost. We had not really any definable code coverage and thus also lost our confidence in our tests. When you are going through some refactoring processes and only half of the code got backed up by unit tests you just don't know anything about the remaining application functionality. At this point it was really hard to raise any motivation for writing tests since they won't provide any value at all.

8.2.3 Catching Up Writing Tests

After the first release we had some time to catch up with unit tests and increase our insufficient test coverage. It did not took long until we realized how much harder it was to write the unit tests afterwards. While implementing one functionality after the other we sometimes completely missed the point of testable design. So we had to refactor and rewrite much code only to be able to write any tests. As this required much time and effort, we were not able to completely catch up with our test coverage. So we tried to focus writing unit tests for important application logic, whereby much untested code was left.

In the ongoing project cycle we were forced to change much of the application functionality because as it turned out it was not going to be accepted by the customer as expected. Every piece of change in our code forced us to update a large set of unit tests. This was especially hard since many of our unit tests became hardly maintainable due to missing experience and a large number of required stub objects. Often it seemed like our unit tests were written to specific since the smallest change in production code broke most of the related tests.

But there were also some good experiences while writing our test-driven application. After some time passed we got better knowledge of writing unit tests and also had an improved understanding of how to correctly use the iOS framework. At least for some specific aspects of the application we were able to write completely test-driven logic, which had

a downright positive impact to its design and also increased our overall motivation to do so.

8.2.4 Conclusion

At the end it didn't looked like our unit testing approach was very successful and it raised the question if the increased effort and costs would provide any value at all. But at least for the sake of the learning process we tried to go on with a test-driven approach for specific application parts when possible.

8.3 Ensure Testable Design

To successfully write unit tests for a whole application, it is important to write completely testable designed code. Most of the design will grow as it should be if development is done in a test first approach.

Overall it is important to always design to an interface instead of a concrete implementation. Only by this it will be possible to replace dependencies by proper fake objects and write good unit tests at all [2, p. 201].

Tell your object what data and object it uses, don't let it ask its environment for this. It is much easier to replace possible dependencies if you decide the location and time where those dependencies get injected. If the object asks for a global singleton instance you have to go through a complex way of method swizzling or override functionality with additional categories. This makes tests complicated and whereby they will be much harder to read and maintain [2, p. 203].

Of course this is not only a guideline for test-driven development but it's important to keep your methods small and focused. Each method should be responsible for exactly one thing, as it should be each test. If a test fails it should be easy to locate the error in the responsible method. For the case when multiple tests fail it should be an easy task to locate the exact location of the error since all of these methods should have something in common [2, p. 204].

Ensure the single responsibility principle for each class. Neither should a class be responsible for more than one thing, nor should a specified task be done by multiple classes. This will remove dependencies and facilitate testing [2, p. 205].

Avoid inheritance. It can be really hard to test tightly coupled classes and nothing is tighter connected then a class which is derived from a parent class [2, p. 208].

8.4 Why Did the Process Fail?

Definitely most of the problems we had were related to the fact that we had quite no experience with iOS development nor had we enough knowledge of a successful test-driven process. It would have helped immensely if there would have been some experience in at least one area of these. Maybe a leader with knowledge in both could have helped guiding us through the process. But in my opinion there is only a minimal chance of an immediate success in these development style if you have that less experience with it.

We temporarily stopped the test-driven development approach after our project was slipped too far behind its schedule. Don't ever stop the process throughout the project or it is going to fail certainly. It does not work writing the tests afterwards no matter how much you believe it will work somehow. Be sure to plan more than enough time from the beginning.

It can be really hard to write unit tests for tightly coupled view and controller logic. Apparently this dependency is enforced by the principle framework application design but it should be tried to comply with the guidelines for testable design as strict as possible. Additionally frameworks like OCMock can help creating stubs for all those framework dependencies.

Bibliography

[1] Osherove R. *The Art of Unit Testing: with Examples in .NET*. Manning Publications, 2009.

[2] Lee G. *Test-Driven iOS Development (Developer's Library)*. Addison-Wesley Professional, 2012.

[3] Patton R. *Software Testing (2nd Edition)*. Sams Publishing, 2005.

[4] Burke EM Coyner BM. *Java Extreme Programming Cookbook*. O'Reilly Media, 2003.

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

[6] Fewster M Graham D. *Software Test Automation*. Addison-Wesley Professional, 1999.

[7] Ken Schwaber JS. The Scrum Guide. https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/2013/Scrum-Guide.pdf. Last accessed: 2014, April.

[8] Kanban - Agile Softwareentwicklung. http://www.it-agile.de/wissen/methoden/kanban/. Last accessed: 2014, April.

[9] Li P. *JIRA 5.2 Essentials*. Packt Publishing, 2013.

[10] Chacon S. *Pro Git (Expert's Voice in Software Development)*. Apress, 2009.

[11] A Successfull Git Branching Model. http://nvie.com/posts/a-successful-git-branching-model/. Last accessed: 2014, April.

[12] Jenkins: An extendable open source continuous integration server. http://jenkins-ci.org/. Last accessed: 2014, April.

[13] OCUnit: iOS Unit-Testing Framework. http://www.sente.ch/software/ocunit/. Last accessed: 2014, April.

[14] GHUnit: Mac OS X and iOS Test Framework. https://github.com/gh-unit/gh-unit. Last accessed: 2014, April.

[15] OCMock: Objective-C Mock-Object Framework. http://ocmock.org/. Last accessed: 2014, April.

[16] Loeliger J McCullough M. *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, 2012.

[17] Conway J, Hillegass A, Keur C. *iOS Programming: The Big Nerd Ranch Guide (4th Edition) (Big Nerd Ranch Guides)*. Big Nerd Ranch Guides, 2014.

[18] Kochan SG. *Programming in Objective-C (5th Edition) (Developer's Library)*. Addison-Wesley Professional, 2012.

[19] Duvall PM, Matyas S, Glover A. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007.

[20] Smart JF. *Jenkins: The Definitive Guide*. O'Reilly Media, 2011.

[21] Freeman S Pryce N. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley Professional, 2009.

[22] Myers GJ, Sandler C, Badgett T. *The Art of Software Testing*. Wiley, 2011.

List of Figures

1.1 A traditional waterfall model 4
1.2 Costs to fix a bug over time 5

3.1 A traditional unit-testing approach 22
3.2 A test-driven development approach 23

6.1 An example of a Jira dashboard 41
6.2 Defining different fields of a Jira issue 42
6.3 Vincent Driessen's GIT branching model 47

7.1 A list of Jira stories 56
7.2 Example of a Kanban board in Jira 58