Achieving High Quality Software for Interactive Mobile Applications

Nikolaus Koller

Master's Thesis

Graz University of Technology

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

Statutory declaration

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

……………………   ………………………………………..

(date)   (signature)

Abstract

This master's thesis is about developing quality software for mobile devices. Smart phones and other devices are part of our daily life. Testing, based on the characteristics of mobile applications, is explained in this document. The importance of testing is beyond dispute, and agile software development methods are becoming more and more requested in an attempt to boost quality and productivity in this field.

Testing and selected agile methods like Kanban and test-driven development are discussed in theory and also as to how they can be applied practically. This document tries to show whether they can help to write quality software or whether it is just hype that makes agile methods popular. The thesis will also consider how these techniques affect the progress of projects and how developers accept the techniques. In addition methods that try to evaluate quality in software during progress are discussed.

In the course of a smart phone project at Graz University of Technology the following topics have been evaluated. The Kanban system has been adjusted to the needs of the project. The rules of the development process have been changed incrementally to find out what would happen. One of the crucial aims of this thesis is to answer the question of how developers can deal with problems concerning new technologies and agile methods.

Table of Contents

1. Introduction

The technological progress of computer science is constantly accelerating. Computers have become smaller and more powerful than they have ever been. The progress of technology has made it possible to shrink high-performance computers to the size of a trouser pocket. New techniques like GPS[1], Bluetooth, Wireless LAN, camera functions along with a comprehensive display have improved the abilities of computers.

This situation has set the foundations for a new challenge. A new market is growing and many companies want to be part of it. It is a place for creative ideas to be realised and a chance for start-up companies to break into this new market. In the last few years, the number of mobile devices has been growing constantly. Smart phones[2] and tablet computers in particular have become part of our daily life. They have the ability to almost replace a personal computer for many people. These changes are also driven by tough competition between companies.

1.1. *The challenge of writing mobile applications*

Modern operating systems provide an easy way to install, remove or distribute new software. This is one of the core features offered by modern systems. It can greatly reduce costs for companies that develop software for mobile devices. Administrative, shipping and advertising-costs can be held down, which allows even start-up companies to get into large markets.

New tools and application programming interfaces[3], often just called APIs, provide a fast and easy way for developers to implement new programs. This also makes it possible to easily use the functionality of new hardware techniques, which expands the potential of every small application. This situation allows companies to release mobile applications with high functionality at a high rate and also boosts markets to grow [AndDev10] [AndSystem10].

These huge markets induce a quickly accelerating development rate of mobile operating systems themselves. Furthermore, the functionality of the operating systems is growing and new versions

---

1 GPS – Global Positioning System is used to find a position on a map.

2 A smart-phone has the functionality of a mobile-phone plus a touch-screen, a powerful hardware and installed software with high ability.

3 An API provides an already implemented functionality to the developer.

are being released frequently. This fast growing brings also disadvantages. A device that runs an older version often does not have the power to run a newer version or it is not worth for the manufacturer to update it. Companies often prefer spending their time and money on new devices with improved hardware specifications (e.g.: New sensors or large screens with higher resolution). This results in fragmentation and makes it harder for developers to write software that easily works on all devices. Most releases are compatible with future versions, but when using the functionality of a newer version many old devices cannot run the application [AndPlat10].

An operating system that can be customized to the manufacturer's needs can bring disadvantages for the software engineer. Many smart phone producers change, for example, the user interface of the operating system. This can have a negative effect on an application that has been developed for the default interface (e.g.: The behaviour of buttons changes). Many devices on the market often show differences concerning hardware, like the size of the screen, battery life, and the power of the processing unit. The different sizes of screens can make a sophisticated user interface difficult to implement. Variations in hardware need to be taken into account when developing mobile applications [Devices10].

The market for mobile applications is a dynamic one and the future seems to be unclear. Many factors have an impact on the quality of software. The goal of this thesis is to show how to develop software for mobile systems with respect to quality and performance. Agile methods are designed to improve quality, but it will never be possible to eliminate all of the problems mentioned above.

Describing all known agile methods would be beyond the scope of this document. This is why the focus will be put on a relatively new lean method named Kanban, which will be combined with test-driven development.

1.2. The properties of modern mobile devices

Many devices with different operating systems have come on the market over the past few years. This thesis is mainly based on a modern operating system for mobile devices named Android[4], developed by Google[5]. It is used for smart phones and tablet computers today. Google explicitly

[4] Android on the web (March, 2011): http://www.android.com/
[5] Google on the web (March, 2011): http://www.google.com/

encourages hardware manufacturers to customize their Android devices with their own more or less complex user interfaces and software add-ons. This of course exacerbates the problems described in the previous section [Gizmodo10].

Modern mobile systems hardly differ from personal computer systems like Windows and Linux distributions. Android is even based on a Linux kernel[6]. Many famous libraries like Libc[7], OpenGL|ES[8] and SQLite[9] are supported. Every application runs its own process within a virtual machine named Dalvik [AndSystem10].

There are two main supported ways to implement an application. The most convenient one is to implement in the Java programming language. This means writing object-oriented software. Java has a garbage collector[10], so there is in theory no need for developers to take care of memory management. The user interface can be defined in either XML[11] or in Java. The other main supported way to implement an application is to write native code. The supported languages are C and C++. When this second way was introduced it was intended for applications that would need more performance, since Java developed applications were slower at that time than native developed ones. An additional reason for introducing the support for developing native code was to import already written software. Drawbacks are code that is more difficult to maintain and the need to recompile code for each new central processing unit (CPU) coming to the market. Both ways of implementation can also be combined. At this point it is also interesting to mention that another technique named Html5 could become a considerable possibility for writing mobile applications [AndSystem10] [AndNDK10].

As we can see, hardware and operating systems are similar to personal computer systems, but there is a significant difference - a hardware limitation, which is due to the small size of the device and its mobility. There is no cable that supplies the device with permanent electricity, thus a battery is needed that also limits the power of hardware. Therefore one major factor is the limitation of

---

6 Kernel is an important main component of an operating system.
7 Libc is a library that provides base functionality in C Programming Language.
8 OpenGL|ES is an API that provides 2D and 3D functionality.
9 SQLite is an engine for relational databases.
10 Garbage-Collector: When allocating memory for data there is no need to free it afterwards. This is done automatically by the Garbage-Collector.
11 XML is an Extensible Markup Language that provides data in a hierarchical structure.

resources, which results in a limitation of memory and CPU[12] computing time.

A limitation of resources has an impact on the life cycle of an application. There is little interest in taking care of a periodical start and stop of an application when developing software for a desktop computer, whereas on mobile applications this is a very important topic to take care of. The operating system may stop or even remove programs from memory when the system runs out of resources [AndDev10].

The graphical user interface is an important topic when developing software. When planning the interaction between the user and the mobile device it is important to be aware of the input methods and their properties. The screen is the main input device on mobile phones, which is usually small and can therefore make a user interaction difficult. Some of the devices have a hardware keyboard, but most mobile phones replace that functionality with a virtual keyboard that appears on the screen when necessary.

1.3. Quality aspects in software engineering

Quality of software can have several definitions depending on the point of view taken. A user of an application wants the software to fulfil his expectations. An engineer wants to have easily maintainable code when doing changes. All parties, when creating software, have their own expectations. Probably there is no way to fulfil all the needs of every person involved, but there are some topics that should, if considered correctly, improve the quality of the software. These topics are: reliability, security, time performance, space performance, portability, maintainability and usability [Spinellis06].

1.3.1. Reliability

Reliability means to have software that provides functionality under pre-determined conditions and that keeps working as intended under unexpected incidents. The functionality, the conditions and likely influences are often defined either directly or indirectly by the requirements of an application, but due to software’s complexity not all factors can be considered.

[12] CPU -Central Processing Unit

When a situation occurs that has not been considered during development a failure might occur. The actual source code that can cause a failure is called faulty. If the application is able to catch the failure, it might be able to function correctly. Otherwise the software state would turn into an error state. Such a state means to have no or lesser functionality provided at this time, depending on the software's behaviour. The way software is prepared to handle problems determines its reliability [Spinellis06] [IEEEGloss90].

Generally speaking, a problem that cannot be dealt with is called a bug and the reliability of software can easily be described with: “The more bugs software has, the less reliable it is”. The reliability of software can theoretically be calculated. The likelihood of a failure appearing during a program execution determines the reliability. The problem with this theory is that it is not feasible to calculate this likelihood. The complexity of a software is too great to predict any misbehaviour and the more functionality is provided the more complex it is [Spinellis06] [UniCarMel99].

A way to be sure that the software works as intended is testing. The high complexity of software makes testing hard. A way to increase reliability would be to keep the software in defined and tested states. Therefore it is necessary to discover potential sources that might have unpredictable influences on the running of a program.

Unpredictable incidents might occur when using external features that are provided by the mobile phone and the operating system. The problem is that such features can have an unclear behaviour in particular situation, because there is often no detailed description. Furthermore, it is not clear what is tested to function correct and under which circumstances those tests were performed.

Further known issues that can have unpredictable effects are the handling of input data as well as interactions carried out by the user. Since those influences can never be fully predicted, implementing and testing needs to be done keenly accurate when developing such features (Monkey testing[13] refers to the way to test unpredictable input data). Interactive mobile applications have a lot of user interactions, which might be a potential source of failures [Spinellis06].

Generally speaking, external interfaces (GUI, system API, etc.) can have a crucial impact on reliability.

[13] Monkey testing means to have no special intention for a test case, but create a random input to test.

1.3.2. Security

Security in software engineering is about hacking and spying to get personal information. An important issue regarding security is that software that is under malicious attack should still be able to function as intended [McGraw04].

There are two different point of view relating to making an application secure. The first one is to consider security during the design phase. Design flaws and incorrect error handling are the main reasons that can lead to security leaks, because hackers might exploit such flaws to harm the software. A thought-out design would prevent the system from getting into an unprotected state [McGraw04].

Security requirements are necessary and they can only be defined when analysing what needs to be protected, from whom and for how long. Many known issues like buffer overflows, race condition, access controls and trust management can be part of the security requirements, depending on the functionality of software, the environment and the programming language. Buffer overflows are a major security topic when implementing in C, but not in Java [McGraw04].

The second point of view considering software security is to protect it by its environment while running. A technique called sandbox protects the software and the system from mutual influence. In the Java programming language, the application runs in a virtual machine that acts like a sandbox [McGraw04].

1.3.3. Space and Time performance

Generally speaking, the time performance of software is based on the algorithm used. An algorithm can be written in many different ways, but lead to the same result. The significant difference lies in the time it takes to finish [Spinellis06].

Special attention has to be paid concerning time performance when using the services of the operating systems. They can, for example, perform a context switch[14], which costs a significant

[14] Context switch: It is a feature of multitasking machines. e.g.: The memory-state of a context (may be a process) is stored and a other one is loaded.

amount of time. Reading or writing to peripherals, like on memory cards, might also slow down the machine [Spinellis06].

As described in section 1.2, memory and power supply are limited on mobile devices. Memory management is influenced by the system, and the developer's influence is rather restricted. On Android it is recommended to be careful when creating new objects. It is advised to re-use old objects to save memory [AndPerf10].

A general rule for creating mobile applications is: Only do what it is necessary to do. When computing memory intensive media files such as images, videos or sound files one should prevent the system from running out of memory. In such cases analysing and developing with respect to space performance can help [Spinellis06] [AndPerf10].

1.3.4. Portability

This topic is about writing software for more than one specialization of operating system and device. It should be possible to port software on other devices or other operating systems without many difficulties [Spinellis06].

Localizing the application to different countries and languages is also part of this topic. On modern mobile systems localisation is often supported by the system itself [AndDev10].

When the operating system supports native code, written in C or C++, source code can often be transferred without many difficulties. Applications that are written in languages like Java have the advantage of being executed in a virtual environment. When the virtual environment is provided by the other system it can be ported easily [Spinellis06] [AndPlat10].

1.3.5. Maintainability

A software development process itself is a constant process of change. Software maintenance is a requirement from the beginning of any software development process until the release and in particular at a later time. Discovered faults or an extension of functionality are the main reasons that may require a change. Significant changes can even affect the whole architecture of a system [Spinellis06] [IEEEGloss90].

Changes to software should be done easily without having a negative side effect, such as creating faults or breaking the rules of the given architectural design. So it is an important requirement for software maintenance that the application stays stable and maintainable after changes have been applied [Spinellis06].

To guarantee maintainability the whole system should be simple and consistent. The architectural design and the source code must be understandable and readable. This makes it easy to introduce new features, make changes and find bugs. Good documentation and test cases help to make successful alterations. Tests would fail if unwanted changes are done.

1.3.6. Usability

Usability is a very important topic in software engineering. An application might work correctly, but if a user cannot understand important parts of it the software can become useless. The quality of the usability of software is evaluated by its ease of use. How easily software can be used can be tested by usability tests. They evaluate the intuitiveness of a user interface and the time it takes a user to finish a task [MS00].

Especially on smart phones, testing the usability is important since the input device is limited. This topic lies outside the scope of this thesis but would be a very interesting topic by itself.

2. Theoretical testing of software

As history shows in many cases software testing should have been applied more accurately in order to save a lot of money, time and stress. Many examples that prove the importance of testing can be found on the web[15].

The traditional software development processes imply that bugs discovered late increase costs. Design faults, for example, might be discovered late, because the faults are often not obvious at the beginning, as not everything can be clear at that time. They often turn out to be faults later on when particular design problems occur (e.g.: Integrating new features is difficult due to software architectural reasons). Therefore a continuous analysis of requirements and testing is really

[15] Famous bugs (March, 2011): http://www5.in.tum.de/~huckle/bugse.html

important. Furthermore from a modern point of view, testing has to be part of every software engineering process and has to take place from the beginning until the end. [AmmannOffutt08] [PezzeYoung08].

When testing software, its functionality is verified against its correctness. The system is put under different circumstances to find out if the situation is handled as expected. Therefore the situation and the desired result need to be defined beforehand [PezzeYoung08].

To gain a better understanding, the software of modern mobile applications can be imagined as a graph of nodes and edges. The state of the software determines the position (the node) in the graph. When for example software has just started, the state "start" would describe the actual position. The state itself has an influence on the software's continuative run. There are also external influences that affect the state and the path a program can take. In every step a program or its components can change into another state and therefore take another path [AmmannOffutt08].

External influences, such as user interactions or file inputs, have a crucial impact on the state of the software. In particular applications on smart phones are designed to have a high user interaction rate. They often trigger the software to execute a path (e.g.: When pressing a button while playing a game).

All those aforementioned factors determine the run of an application and the path a program can take. External influences have to be considered carefully, because their occurrence is often unexpected (e.g.: A user touches the screen without particular reason). To ensure that such a complex structure works as it should, testing is required [AmmannOffutt08] [PezzeYoung08].

It is not possible to test the complete software in all its possible states, since that would even in small cases take millions of years. Testing is not a way to completely avoid faults. Testing cannot show the absence of failures, it is more a way to illustrate that particular functionalities work correctly. The functionality that is defined by the requirements of an application can for example be taken to create test cases. This is called functional testing [AmmannOffutt08] [PezzeYoung08].

There are different methods available that can help to satisfy the requirements of testing. Assessing which method is the best depends on the functionality and requirements given by an application [AmmannOffutt08] [PezzeYoung08].

2.1. *Testing methods*

Test cases can never guarantee a fully correct working program execution, as described in the section above, as it would be necessary to consider all states and steps a program can take and test them. When looking at software testing in a realistic way, the case of testing needs to be divided into different methods and criteria. Because it is infeasible to test the complete system, a method that fits to the particular situation needs to be applied [AmmannOffutt08].

Coverage criteria is a newer testing terminology than the testing methods described in this chapter. In this project there are many new topics like test-driven development and Kanban as well as the relatively new platform, Android. Exploring coverage criteria is outside the scope of this work, thus the older well-established terminology is used [AmmannOffutt08].

An explanation of coverage criteria can be found in: ”Introduction To Software Testing” written by Paul Ammann and Jeff Offutt.

There are two kinds of testing methods that can help to satisfy the given requirements of testing. They differ in the way one looks at software. On the one hand the software can be seen as a closed system also called a black box. Tests that derive from this point of view are also called functional tests, since they test the working of defined functionality. On the other hand it is possible to test the software according to the inside of the box called the structure of software. It is also called white box testing. Within mobile applications a combination of both methods can bring benefits. This is discussed later more in detail [AmmannOffutt08] [PezzeYoung08].

2.1.1. Functional testing / Black box testing

When applying black box testing the focus is on the input and the output data of software. Most input and output data should already be defined within the specification of an application. The ability with which the computer can compute the output is called functionality. This plays a crucial role in black box testing [PezzeYoung08].

To prove the correct functionality the resulting output has to be compared to the expected one. A variance in these two values would state a fault. A simple functional test might already show the partial correctness of a large part of the application, which can make testing very fast and simple.

However there is a crucial disadvantage in writing functional test cases. If a test case fails, it is very difficult to find the fault, since there is no detailed information about the particular faulty part of the source code [PezzeYoung08].

2.1.2. Structural testing / White box testing

When applying structural (white box) testing the software is tested according to its structure. The application's specification itself is not directly considered within these test cases, like it would be when writing functional tests [PezzeYoung08].

Software consists of many different components like interfaces, classes, libraries and further constituent parts. Every part of the software has its own function within a system. When these components interact with each other, every part has to rely on the correct functionality of the other one. Structural tests verify if these parts and their interactions are working as intended [PezzeYoung08].

When implementing structural test cases a benefit is that faults can be found faster than in source code that is covered by functional tests. When a structural test case fails, the location of the fault can be determined, since the code covered by this particular test case is faulty or at least gives information on where to find the mistake.

Generally speaking structural testing needs more work than functional testing, but brings the advantage of fast fault detection.

2.2. Testing process levels in object-oriented software

In modern mobile systems object-oriented programming languages are usual. When testing object-oriented software it is important to consider its paradigm. This paradigm includes aspects like polymorphism, encapsulation, abstraction and inheritance. Exception handling is not part of that paradigm, but seems to be getting more important. These aspects need special attention when writing test cases, because they can cause faults as a result of their complexity [PezzeYoung08] [AmmannOffutt08].

Inheritance

Inheritance allows a class to derive from another one, which means grouping objects that have something in common. This brings the benefit of implementing particular parts of code just once and also an advantage when designing architectural structure. The inherited class can also be extended by new functionality, or inherited methods can be overwritten. The behaviour of an inherited object can differ from the behaviour of the parent-class. Therefore it is important to consider writing new test cases and ensuring if the old test cases that were created for the parent-class still satisfy the testing requirements [PezzeYoung08].

Polymorphism

Polymorphism makes it possible to change the type of a variable at run-time. The type of variable can be changed into one that has the same root of a class in its hierarchical order of types (A technique that makes it possible to achieve such a hierarchical order is inheritance). Polymorphism means to have influence on the behaviour of a system at runtime. Therefore special attention is necessary when writing test cases [PezzeYoung08].

Abstraction

Abstract classes cannot be tested directly, because there is no instance of a class that can be created. Nevertheless those classes define an interface to a component that might be used and should therefore be considered when writing tests. Furthermore all likely subclasses can be created and tested to examine the correctness of their behaviour [PezzeYoung08].

Exception handling

In modern programming languages exception handling can be used to handle particular situations. It is not part of the object-oriented paradigm, as already mentioned, but it can have a crucial influence on the reliability of a system. When an exception is thrown up the handling of that special situation is responsible for the system's new state, so it can help to handle unexpected failures and to maintain a stable state. Furthermore, exception handling may play an important role concerning quality of code [PezzeYoung08].

Encapsulation

Encapsulation is important to keep a class or a component safe from unwanted influence on an object's state, thus not any method of a class can be accessed from the outside. Methods can be declared by different keywords that define the rules of accessibility. Some of them can only be accessed from the inside of a class and some of them from the inside as well as the outside. These rules and possibilities can vary in different programming languages (e.g.: Keywords like public, protected and private are usual in the Java programming languages). When writing test cases it is important not to break the rules of encapsulation. Changing such a keyword in order to write tests would break the rules.

Object-oriented software is condition-based. Each object can turn into a state that is mainly influenced by its methods and parameters. The result of a method is also influenced by an object's state. Therefore a class should be tested as a unit, because the test result might depend on the object’s state.

Testing such a unit would be the lowest level in testing object-oriented software. Testing the interaction of that class would be the next highest level, known as integration testing, followed by acceptance (system) testing, which is the highest level of testing object-oriented software [PezzeYoung08].

2.2.2. Intraclass testing / Unit testing

As already described in the section above the smallest unit that should be tested in object-oriented software is a class or rather an instance of a class (also called an object) and its state. An object has to be seen as an inseparable entity that plays a particular role in a system and should therefore be tested as a unit [PezzeYoung08].

It is important to define the states an instance can turn into. A state chart could help to keep an overview of several states. In this case it is necessary to gain information about the possible states and the triggers that force an object to change from one state to another. According to that information intraclass tests can be written [PezzeYoung08].

2.2.3. Interclass testing / Integration testing

Integration testing can be much more complicated than testing a class' behaviour itself, since the software's structure gets bigger with every integrated class, this also has an influence on the complexity of a structure. So it is important to find a strategy that guides someone when integrating classes [PezzeYoung08].

A Bottom-up strategy might be the first step when starting to write interclass tests. In this case someone has to start integrating classes that have no dependencies and then move onto entities that have one. Hence, just a small group of classes can be considered at first. Step-by-step more classes need to be included afterwards. Abstracted or inherited classes are not important, but their implementations are. Modelling objects into a diagram, like a sequence diagram or a collaboration diagram[16], helps to discover class interactions. A sequence diagram simply shows what should happen next in the software's run. A collaboration diagram describes, as it says, the collaboration between the classes [PezzeYoung08].

2.2.4. System testing / Acceptance testing

System (acceptance) testing is the final testing phase in a software development process. In this phase the required specification is proved to be implemented. The aspects of quality in software may even be part of that testing period [PezzeYoung08] [IEEEGloss90].

2.3. Regression testing

When a system is extended or maintained, it is tested again to ensure that no unwanted change has been created. New functionality has to be tested as well as the old one. When a bug is discovered and fixed, regression testing also involves the writing of a test case to prove the absence of the bug [IEEEGloss90].

In test-driven development, which is discussed later in detail, a regression test is written **before** fixing the code in order to ensure that the test case correctly covers the bug so that its later re-appearance (due, e.g., to copy paste from older code) is not overseen.

[16] More informations concerning sequence diagrams or collaboration diagrams can be found on web.

3. Agile software development

Making quality software, which is maintainable, reliable and that has a predictable lead time are the goals of a modern software development team. Furthermore increasing social value and skills should also be part of this goal. Every development situation is different. It can differ in team size, team members, social value, skills, company hierarchy and structure as well as the type of software that needs to be created or maintained. It is difficult to define hard and fast rules that guarantee creating quality software in every situation. Agile software development methods can be applied to deal with all these issues. They are lean and agile and should therefore fit into almost every situation [Anderson10].

Agile methods are relatively new to most software companies, but are getting more important every day. Those methods help to implement software in a flexible way, which should lower administrative work and reduce dependencies. A dependency can be given when a piece of code is written by a developer in a less comprehensible way. When this developer leaves for whatever reason, there might be a lot of work to introduce another developer to the given code. Agile methods are applied to lower that risk.

Many agile methods have been developed over the last years. The most famous ones are: Extreme programming (XP), Scrum, Kanban, test-driven development, feature-driven development and program-driven development [Devagile10].

When introducing an agile method in a company it is important to do this carefully with respect to the participating persons, because people might be afraid of changes. There are agile methods that have significant influence on a team's structure like Scrum and Kanban. A thought-through introduction helps to increase the acceptance of change [Anderson10].

For the project that will be described later in this thesis it is important to apply a method that helps to administer the development process and one that makes the source code highly maintainable. This document describes Kanban as an agile method in combination with test-driven development, since they promise to satisfy these requirements. They should help to achieve a high quality code [Anderson10] [Beck03].

3.1. Agile tests as documentation

Usually technical documentation is a textual description but it can also be done by writing test cases. There are many advantages of having tests as documentation compared to a textual one.

A lot of written text would be necessary to describe functionality in detail. A developer would need to spend a lot of time to understand the source code's intent. Test cases can define in detail what source code is intended to do and engineers might even better understand a test case than a written passage.

It is a lot of work to write test cases and to keep them up to date as well as writing a textual description, but when testing is applied the documentation would already have been done. When the software is modified and the tests are adjusted to satisfy the testing requirements, the documentation is automatically kept up to date and there is no extra work. In contrast, traditional textual documentation is often neglected because other activities like adding additional functionality are accorded higher priority by engineers, the reason of often being either bad planning or individual preferences of the engineers – many technically oriented people simply like coding more than writing documentation they feel is redundant and thus a waste of time.

An additional advantage is that tests prevent the source code from unwanted modifications, since test cases would fail if there was a change that is in conflict with the specification. In a textual description it might be overlooked. So tests as documentation allow automated protection from unwanted changes.

Nevertheless when using this method a high test coverage that is continuously kept up to date is really important, otherwise there would be no detailed documentation.

3.2. Test-driven development

Test-driven development means writing a test before implementing the actual source code. This is applied to every piece of code. Test-driven development has the following advantages:

- What should be done is written down in keywords or short sentences. Then a test is written that checks one of those issues. The test case defines what to do. This might help to keep the focus on the actual task and therefore should help to implement a simple encapsulating design.

- The test cases illustrate the already implemented functionality. So it is not necessary to keep a detailed overview in mind about what has already been done and what remains.

- Achieving high test coverage is a main advantage of applying test-driven development. When test-driven development is not applied the test cases might be written after the functionality is implemented. Then there is little interest in obtaining high code coverage, since there is often little time left. This situation constrains companies to write a satisfying number of test cases. So test-driven development makes testing a constant part of the whole software development process.

- In a minimal but effective way the documentation is already completed by writing test cases as described in a previous section.

- High testing code coverage offers protection when changing source code. If it is necessary to alter a given source code that has high code coverage the developer feels safe when performing changes. If a mistake is made, test cases would show it. So the loss of already implemented functionality is prevented.

[Beck03]

The following instructions demonstrate the basic test-driven development cycle:

1. Choose a task and analyse the situation. When everything is clear define a goal.

2. Take that goal and split it into tasks. Write them down as keywords or short sentences.

3. Write a test that proves one of the issues and therefore defines what to do. This test should fail, since there is still no code of functionality that fulfils this test. Do not create too many tests before writing the actual code otherwise a long period of failing tests would follow, which might lower the developer's motivation. Some tests might also not be relevant anymore due to decisions taken while writing the code corresponding to the tests.

4. In the next step the actual code is written until the test runs green. All of the other older tests should pass too. New issues can arise during development. Add them to the list of issues,

except if they are absolutely necessary to be implemented immediately.

5. Refactor the source code. Within this step remove unnecessary lines, comments and duplications of code and adjust given names when they are misleading. This step increases the readability.
6. Go back to step (3) and go on until the task is done.

[Beck03]

There are no strict rules on how big the steps of writing test cases and code are. The amount of work that is done within one circle should depend on how certain a developer feels within the particular requirement. Taking small steps is recommended when refactoring unfamiliar source code or developing in a field one has no experience in [Beck03].

Implementing new features or maintaining the actual source code is the software developer's daily business. Adding new functionality often requires maintenance as well. In both cases, the cycle described above should be applied. Even when writing regression tests one should act like this [Beck03].

Further information can be found in the book “Test-Driven Development” by Kent Beck. Examples are described as well as several patterns that help to apply test-driven development in particular situations.

3.3. Kanban

Kanban, created in Asia and applied in process administration (e.g.: In car manufacturers), is adopted into information technology as an agile software development method. Kanban in IT is used to keep an overview of the progress of a project and should make bottlenecks and other kind of problems visible. Furthermore Kanban should make it possible to easily estimate lead-times and to prevent unexpected delays.

A board is used to keep an overview of the project's tasks. Each ticket on the board symbolizes a task, and its position shows its progress. The board consists of several lanes that define a particular phase in a task's progress. When a ticket finishes a phase it is moved into the next lane until it is finished. Usually an amount of tickets is created that can be processed in a particular period of time. This should help to stay agile and to keep the actual work in mind. At least there are no rules on how big a period of time should be.

Fig. 1 describes a simple Kanban board.

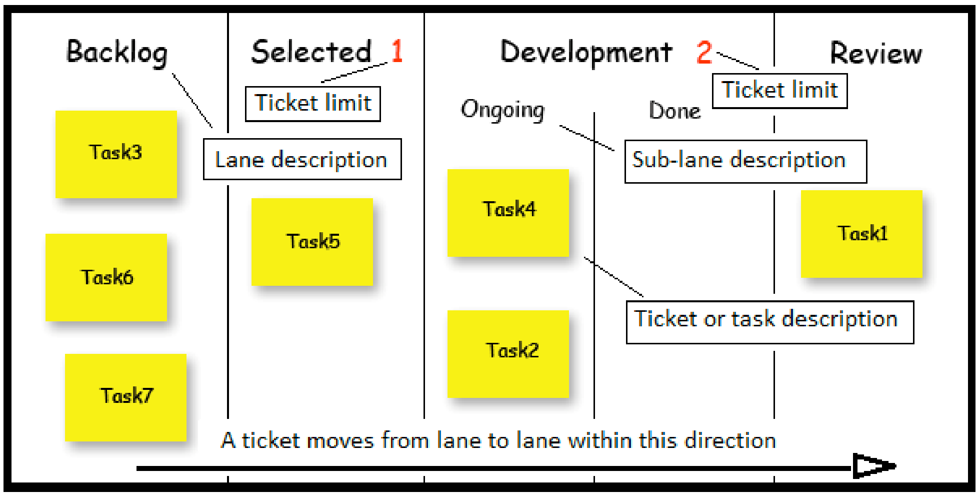


Fig. 1: Kanban board description

Kanban can best be described by its attributes, which are the following:

The pull-system

One of the principles of Kanban is the pull-system. Mental pressure on a developer is decreased by enabling him to pull a new task when the assigned one is finished. Too much pressure on a developer might lead to a sloppily finished task, which can include mistakes and low quality code. Such problems need further work, since it is the goal to achieve high reliability and highly maintainable code. Furthermore, this can lead to faults that will appear later in progress, because this part of code can be a basis for another task that might be affected. Hence, the task is not finished because there is still work to do. This principle of Kanban is supported by the board, while a ticket can only be moved (or pulled) forward if there is space in the next lane [Anderson10].

The work in progress limitation

A team has a limit to the capacity of work that can be done at any one time. Work in progress is limited in order to prevent exceeding this capacity. The limitation depends on the team size and the tasks that are performed. Usually the limitation of tasks equals the number of entities in a team. An entity can be a single developer or more (e.g.: In pair programming an entity would consist of two developers). The number of entities should be increased by a small buffer. This buffer is important to ensure progress when some tasks are stuck. The buffer allows tasks to be kept in a particular lane for a while. There might be several reasons for having blocking tasks like waiting for a specialist who is familiar with a particular problem [Anderson10].

The work in progress limit is not a static number. It needs to be adjusted during progress according to the current circumstances. When the next task needs to be scheduled, the team has to discuss that decision since the number of actual executed tickets is limited. The discussion forces a better understanding of the overall situation. Every developer is kept up to date, which results in better productivity and lead time [Anderson10].

In Fig. 1 the limit is shown beneath the name of a lane. It is important to limit lanes in which bottlenecks are likely. The subsequent lanes can be limited but it is not necessary, because the limitation in the previous lane influences all of the subsequent ones. As is shown in Fig. 1 the development lane is the most likely bottleneck lane in this particular case [Anderson10].

In the backlog lane the number of tickets should be restricted as well. Usually the number of tickets equals the number of tasks that can be done in a period of time. This number is often not certain at the beginning, but should become clear after a while [Anderson10].

Visualizing the work flow

One of the major goals in Kanban is to visualize the work in progress. A Kanban board is used to give an overview, as shown in Fig. 1. Such a board is arranged into different lanes. Usually there is a backlog lane, a selected lane, a develop lane and some kind of deploy lane or review lane. In the backlog lane are some tasks that are expected to be finished in a period of time. In the selected lane are the tasks that should be done next. The develop lane shows the active development jobs and the deploy lane shows the tasks that need to be reviewed and integrated. How to build up a Kanban board depends on the particular situation. There are no strict rules as to how the lanes should be named or how many of them are necessary [Anderson10].

It is important to split some lanes apart. A ticket is moved into the first sub lane. After finishing the task the ticket is moved into the second part. Now the ticket is ready for the next step and can be pulled into the subsequent lane, which is important to ensure the pull-system principle. The lanes at the beginning and at the end can also be sub divided, as desired [Anderson10].

A piece of paper, called a ticket, is pinned on the board in order to show the progress of a task. The Kanban board technique can simply give an overview of work and progress, but it is not intended for a team that is distributed over several places. In this case a software solution can be used to visualize the progress. This can either be used alone or combined with a real board in a physical location. Someone would need to keep the real board up to date [Anderson10].

Tickets

A ticket describes a task like a requirement or a bug that should be fixed. There is no special rule for the content of a ticket. It makes sense to assign an incremental number to each ticket, because following a task is then easier. Writing down information like the assigned developer and the amount of work can help administrate work. The value of priority on a ticket can help to determine the task that should be started next.

The prioritization

The tasks are ranked according their current priority, which means that the system makes it possible to start a task immediately after it has been added. A discussion of the importance of particular tickets is necessary to evaluate the priority of tasks, which ensures a clear direction in a project's progress. Tickets that are not processed within a period should be deleted and not considered in the next period, because they would only reduce the agility. If they become important later in the process they can be re-defined under the new situation [Anderson10].

The number of tasks that should be assigned to a period depends on the project's circumstances. It cannot be defined by a static number, but has to be part of an empirical process [Anderson10].

The Kaizen culture

A lean and agile method like Kanban needs to have a special atmosphere in a team to ensure success. Kaizen means continuous improvement in quality. A Kaizen culture has a high social value. Everyone regardless of position in the hierarchy feels free to make changes. They help each other and try to improve quality. Developing a Kaizen culture should be aimed for when trying to use Kanban successfully [Anderson10].

4. Practical application

The main goal in practical application is to reach a high level of quality. One important challenge is to uncover quality faults during the development of an application. The applied agile development methods will be adjusted according to the problems found. Test-driven development, tests as documentation and Kanban are applied within the project to reach the aspired goal. These methods will get evaluated in order to find out if they have an influence on the quality of software and how they can be applied successfully.

In this chapter, the initial situation is analysed, since it has a crucial impact on the methods applied. During the application the project's progress is monitored and problems are discovered. At the end of this chapter, the discovered facts are analysed and explained.

The application

The project named 'Catroid' is intended for children to get a first experience in software development. It is a graphical programming language that allows building small interactive games on mobile devices. The project is kept open source. Details about the project as well as the application’s install file (.apk) can be found on the web[17] for download.

[17] Catroid project page on (March, 2011): http://code.google.com/p/catroid/

The requirement of quality

The main focused quality aspects will be maintainability, reliability and performance.

Maintainability is probably one of the major goals an open source project should have, since many different kinds of people all over the world should be able to work on the code without the possibility for face to face communication.

Reliability and performance is crucial to make the software acceptable to the public, otherwise there would be no interest in using the software in a serious way.

Portability is a desired feature, but is not intended to be a main goal in this project.

So far it is intended to use the Java programming language, which ensures a basic level of security. The project does not need any further security measures, while there is no interest in making it acutely safe.

The usability topic would be of course one of the major ones, but it is not considered within this document which concentrates on the agile software development process and leaves out all usability techniques applied in software development processes.

4.2. The initial situation

The actual situation has to be analysed, because it has a crucial impact on the result of this thesis. The team, especially its motivation and skills, as well as the time they can spend are essential values that influence the project's result. Being confronted with new methods and technologies should be considered when evaluating the output. Those topics are analysed within the next sections.

4.2.1. The team

The planning and programming is performed by students from Graz University of Technology. Guidance and support is done by the Institute of Software Technology. The work by the students is not paid but credited for university lectures like the bachelor or Master’s theses. Hence, money as a standard motivation factor does not play a role. The major driving factors are one’s interest in mobile application development for the children of the world using agile methods in an open source free software development project.

The team is very dynamic, since there are no special rules as to when to work or where. A room is provided in which the developers have the possibility to work, meet and discuss issues. It is also the place in which the Kanban board is managed.

During the project students will quit (after finishing their work) and new ones will start. Hence, knowledge acquisition can be difficult. It is likely that most of them have no or little skills in mobile application development and Kanban, because those topics are fairly new.

The biggest challenge concerning the team is to keep up the motivation in order to make them accessible for agile methods and its rules. Otherwise it would be difficult to apply the techniques successfully and therefore achieve high quality software. Furthermore, the evaluation of the agile methods could fail.

4.2.2. The technical environment

A short illustration concerning testing and the technical environment of mobile applications are given in this section. It is necessary to successfully use the features provided and understand its techniques. Furthermore, in this chapter I discuss how source code of an Android application should be handled to write useful test cases. Some software sections will turn out to be easier to test than others. As a result some kind of classifying source code is necessary. Therefore it is important to consider testing as well as the particular technical environment when planning the application's architectural design.

Testing a mobile application's source code

When testing, the source code of an Android application can be classified into two kinds. One kind that has a low Android specific source code quota can be determined by its minor use of Android APIs. In this case testing is done as is usual in Java applications. Particular problems that need special attention have already been described in the chapter about testing.

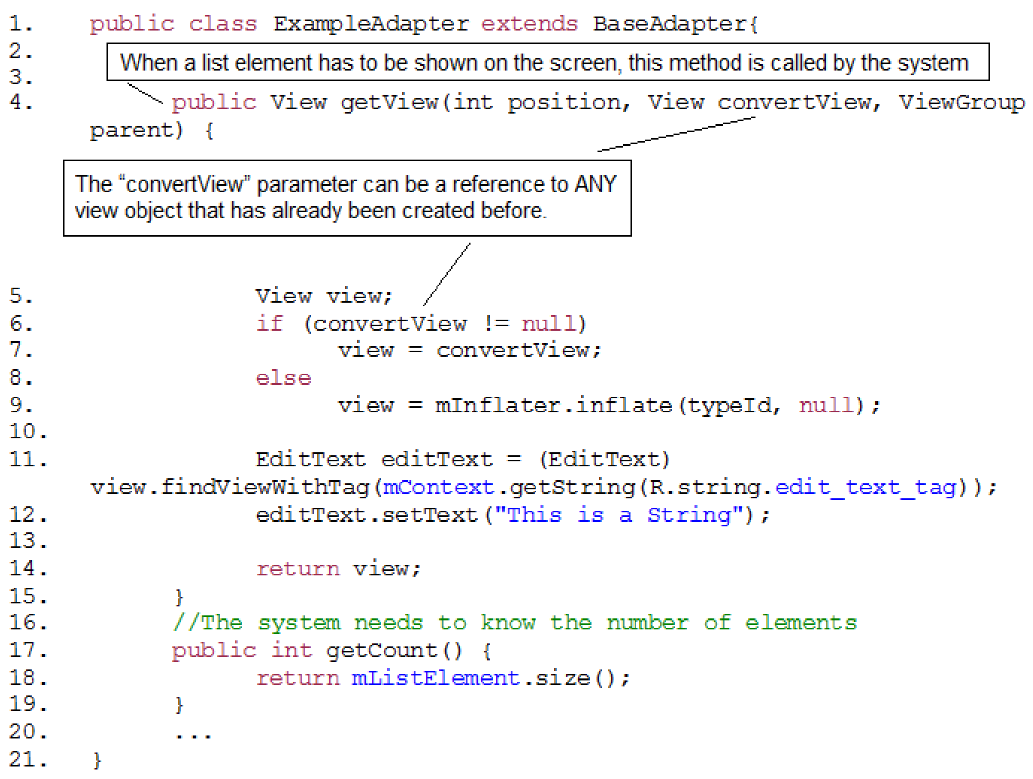
The other kind is determined by a high quota of Android specific source code. This code provides functionality that allows an easy handling of GUI interactions and communications to the mobile phone's hardware elements and other features.

When testing Android specific source code two major difficulties need special attention:

- The tight integration with one’s own source code
- The variation in result when using Android features on different devices

The tight integration with one’s own source code#

Many Android features, especially GUI functionalities, are tight integrated into one’s own source



```
public class ExampleAdapter extends BaseAdapter{

          public View getView(int position, View convertView, ViewGroup
parent) {
                View view;
                if (convertView != null)
                      view = convertView;
                else
                      view = mInflater.inflate(typeId, null);

                EditText editText = (EditText)
view.findViewWithTag(mContext.getString(R.string.edit_text_tag));
                editText.setText("This is a String");

                return view;
          }
          //The system needs to know the number of elements
          public int getCount() {
                return mListElement.size();
          }
          ...
}
```

When a list element has to be shown on the screen, this method is called by the system

The “convertView” parameter can be a reference to ANY view object that has already been created before.

Fig. 2: Shows the tight integration of one’s own code into the system code.

code, which limits the possibility of testing.

In Fig. 2 above an adapter class provided by the system is shown. This feature manages to show particular view elements in a parent-view[18]. The developer has little work to do to make it run. This makes software developing fast, but it is also a loss of control which is demonstrated by the following example:

On every change of content the view elements have to be refreshed to show the actual data. A memory reference to an element object of a parent-view can get lost in a refreshing action, as was found out during this work within the current Android version (Android version 2.2; March, 2011). The Android system changes views and its memory references because of its memory management system. Testing particular objects of a parent-view can be difficult when Android changes the object's reference at arbitrary points of time.

Testing an adapter class' functionality within structural tests is almost impossible, because there is, within a standard situation[19], no access to its source code. Therefore the testing possibilities are limited. Writing functional tests, that cover the code from the beginning of a user action until the result is shown, are a working solution. The adapter class is one of many examples that illustrate the importance of functional testing in Android.

The input data/method that is necessary to start a GUI based functional test has to be a virtual screen action (e.g.: A touch) that triggers the software to execute a particular function. The Android platform provides functionality to do so, but it is still not technically mature.

There is another possibility to simulate user interactions by a tool named Robotium[20]. It is based on the provided functionality of Android but is far more sophisticated.

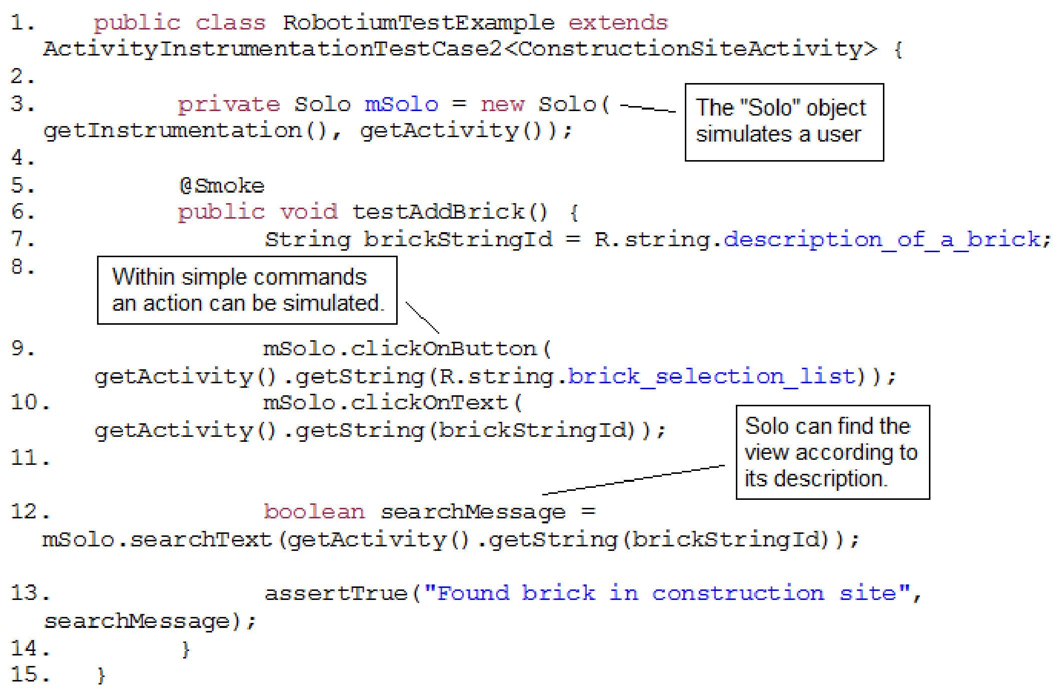
A functional test in Robotium means that a simulated action induces a result to be displayed that can be compared to an expected one. All the functionality that is necessary to perform the expected output is part of this test.

The following Fig. 3 illustrates a Robotium test.

[18] A parent-view provides several elements a place to be shown on the screen in a particular form (e.g.: list-, grid-view elements).

[19] The source code is kept open source, which could make access possible.

[20] Robotium located at (March, 2011) http://code.google.com/p/robotium/



```java
public class RobotiumTestExample extends ActivityInstrumentationTestCase2<ConstructionSiteActivity> {

        private Solo mSolo = new Solo( getInstrumentation(), getActivity());

        @Smoke
        public void testAddBrick() {
                String brickStringId = R.string.description_of_a_brick;

                mSolo.clickOnButton( getActivity().getString(R.string.brick_selection_list));
                mSolo.clickOnText( getActivity().getString(brickStringId));

                boolean searchMessage = mSolo.searchText(getActivity().getString(brickStringId));

                assertTrue("Found brick in construction site", searchMessage);
        }
}
```

The "Solo" object simulates a user
Within simple commands an action can be simulated.
Solo can find the view according to its description.

Fig. 3: Example of using Robotium.

Using Robotium it is possible to click, search, touch and apply many more actions on every kind of view, which gives the developer many possibilities to write functional tests for GUI and other Android platform features.

The variation in results when using Android features on different devices

The second Android specific difficulty when testing is the varying results when communicating with particular hardware or even when using particular GUI elements (e.g.: HTC sense defines its own attributes of views, like the minimum size for the Edit text-view element, as has been discovered within this project).

The fact that manufacturer of mobile phones can alter the operating system to their individual needs, can lead to different results when using particular features. A problem that was found in this work is that some hardware elements behave in different ways, like when trying to control the brightness of the screen as the example in Fig. 4 shows.

Fig. 4 shows a simple Android Application that should toggle light from a low lighted screen to a higher lighted one:

```java
public class ToggleDisplayActivity extends Activity {

   private WindowManager.LayoutParams mWinParams;

    @Override
    public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      this.requestWindowFeature(Window.FEATURE_NO_TITLE);

      getWindow().setFlags(
                  WindowManager.LayoutParams.FLAG_FULLSCREEN,
            WindowManager.LayoutParams.FLAG_FULLSCREEN );

      getWindow().setFlags(
                  WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON,
            WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);

            setContentView(R.layout.main);

            mWinParams = getWindow().getAttributes();

            mWinParams.buttonBrightness =
WindowManager.LayoutParams.BRIGHTNESS_OVERRIDE_OFF;

            mWinParams.screenBrightness = 0.004f;
            getWindow().setAttributes(mWinParams);
    }
```

This value can be between 0 and 1.

This method changes the screen brightness value on a screen touch.

```java
    public void onTouchScreen (View v) {
```

Value 0 means a black screen and value 1 means a screen as bright as possible. When increasing the value from 0 up to 1, step by step, different devices turn on the screen within different values.

```java
      if(mWinParams.screenBrightness == 0.004f)
            mWinParams.screenBrightness = 0.5f;
      else
            mWinParams.screenBrightness = 0.004f;

            getWindow().setAttributes(mWinParams);
      }
}
```

Precisely, 0.004f is the minimal value to light up the screen of a Nexus one. e.g.: The Samsung Galaxy Tab's display still stays dark at the same value.

Fig. 4: An example of a varying hardware result. The application (.apk) file can be downloaded on the web [AndMarket11].

In Fig. 4 (nkNightClock application can be found on the web[21]) the result of setting the screen brightness varies on different devices within the same values. There are devices that already lighten

[21] NkNightClock on Android Market (March, 2011): https://market.android.com/details?id=at.nk.nightClock

the screen and others that may stay dark at the same value [AndMarket11]. Such variations constrain a developer when writing automated tests because functionality has to be proved manually to ensure correct working.

Hardware features like flashing a LED light or starting the mobile phone's vibration function may also vary in result and have therefore to be tested manually.

4.3. Sampling and monitoring phase

Since no special Kanban experience had been developed previously to the project, the project started with a simple Kanban board. The board layout was adjusted continuously to adapt it to our situation.

4.3.1. Introducing Kanban and its process of development

Issues that need to be considered from the beginning are the number of developers that are assigned to the project and the development time they can spend, since one can state that these factors do have an impact on the work in progress and the limitations that should be determined for the lanes of the board.

Another crucial factor influencing the Kanban board will be the quality of code. Its value will be evaluated during the whole project. Gaining this information and adjusting the Kanban board is going to be the biggest challenge within this work. The aim is to discover quality faults by talking to the developers, by reviewing source code and by giving an overview of the project's progress.

The analysing of Mercurial entries will be an important method to monitor the progress. The most noticeable problems are considered when adjusting the Kanban board. The effectiveness of the adjustment will be evaluated. Depending on the result these changes will be kept or rejected. Too many changes at once are to be avoided, because they could distort the result and it would not be clear which particular change led to the actual situation.

4.3.2. Adjustment on 2010-07-07

Start of the Kanban process in the Catroid project. The developers follow the rules given by the

Kanban board. They are advised to focus on quality and trying to develop within the principles of test-driven development.

A simple Kanban board is used as shown in Fig. 5.

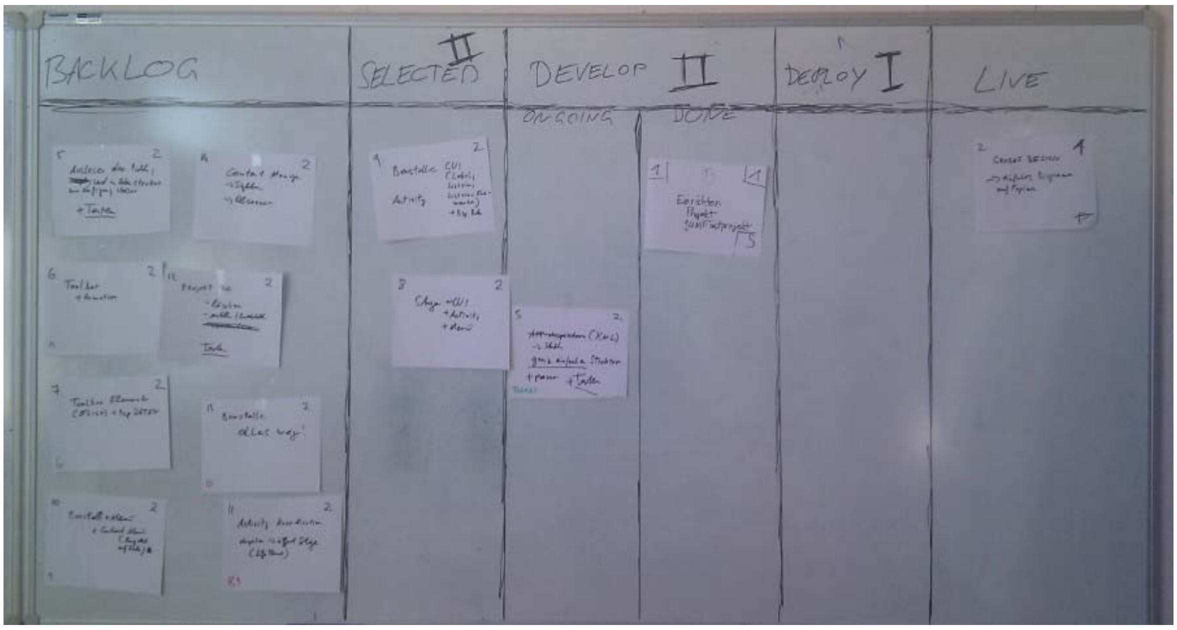
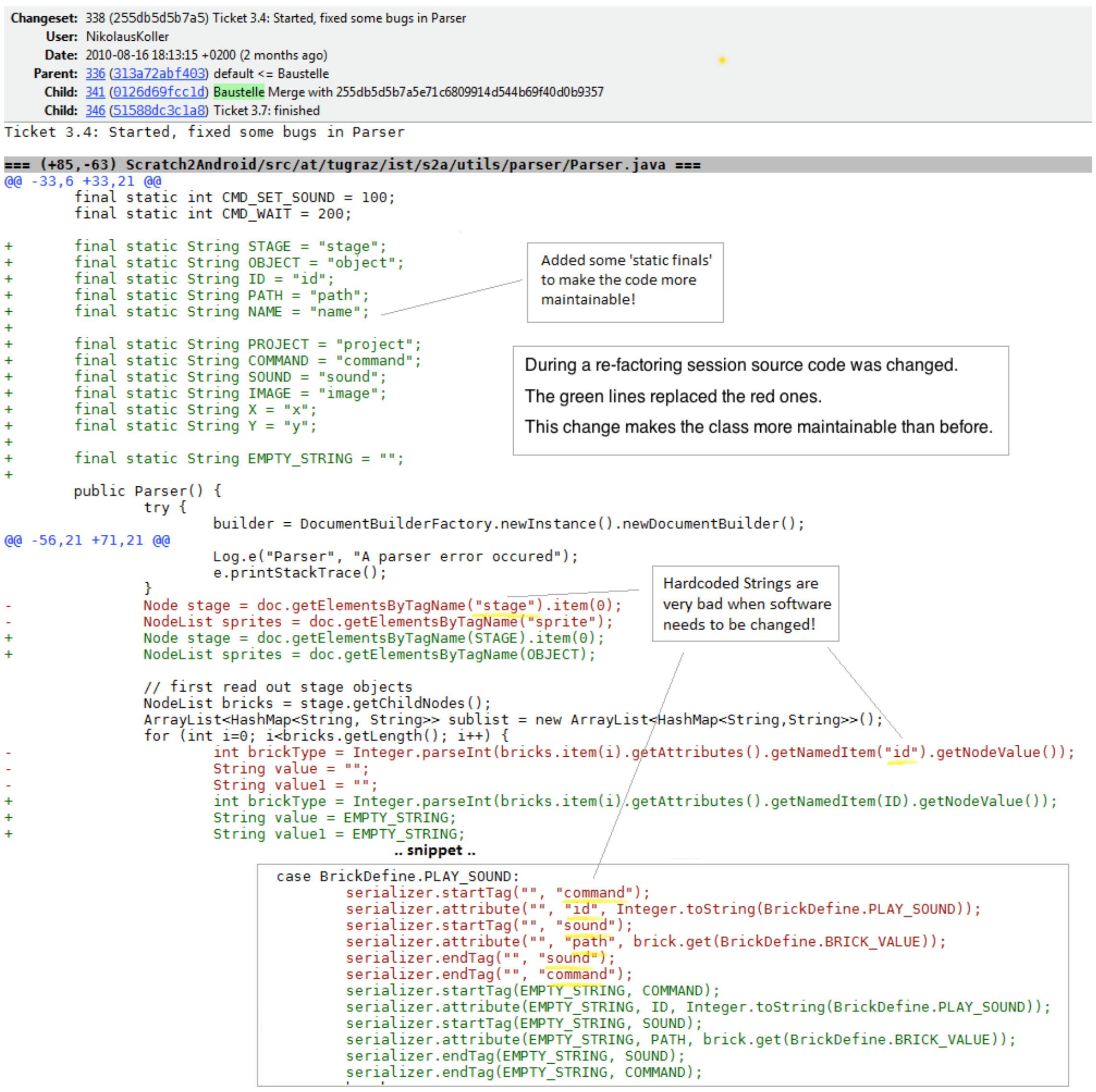


Fig. 5: Picture of the Kanban board from the period starting on 2010-07-08

4.3.3. Adjustment on 2010-09-16

The most noticeable problems in the last period were a less maintainable and a hard extendable source code. Many failures at the end of the period reduced the reliability. A screen shot of a Mercurial page shown in Fig. 6 demonstrates that evaluation.



Changeset: 338 (255db5d5b7a5) Ticket 3.4: Started, fixed some bugs in Parser
User: NikolausKoller
Date: 2010-08-16 18:13:15 +0200 (2 months ago)
Parent: 336 (313a72abf403) default <= Baustelle
Child: 341 (0126d69fcc1d) Baustelle Merge with 255db5d5b7a5e71c6809914d544b69f40d0b9357
Child: 346 (51588dc3c1a8) Ticket 3.7: finished

```
Ticket 3.4: Started, fixed some bugs in Parser

=== (+85,-63) Scratch2Android/src/at/tugraz/ist/s2a/utils/parser/Parser.java ===
@@ -33,6 +33,21 @@
         final static int CMD_SET_SOUND = 100;
         final static int CMD_WAIT = 200;

+        final static String STAGE = "stage";
+        final static String OBJECT = "object";
+        final static String ID = "id";
+        final static String PATH = "path";
+        final static String NAME = "name";
+
+        final static String PROJECT = "project";
+        final static String COMMAND = "command";
+        final static String SOUND = "sound";
+        final static String IMAGE = "image";
+        final static String X = "x";
+        final static String Y = "y";
+
+        final static String EMPTY_STRING = "";
+
         public Parser() {
                 try {
                         builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
@@ -56,21 +71,21 @@
                         Log.e("Parser", "A parser error occured");
                         e.printStackTrace();
                 }
-                Node stage = doc.getElementsByTagName("stage").item(0);
-                NodeList sprites = doc.getElementsByTagName("sprite");
+                Node stage = doc.getElementsByTagName(STAGE).item(0);
+                NodeList sprites = doc.getElementsByTagName(OBJECT);

                 // first read out stage objects
                 NodeList bricks = stage.getChildNodes();
                 ArrayList<HashMap<String, String>> sublist = new ArrayList<HashMap<String,String>>();
                 for (int i=0; i<bricks.getLength(); i++) {
-                        int brickType = Integer.parseInt(bricks.item(i).getAttributes().getNamedItem("id").getNodeValue());
-                        String value = "";
-                        String value1 = "";
+                        int brickType = Integer.parseInt(bricks.item(i).getAttributes().getNamedItem(ID).getNodeValue());
+                        String value = EMPTY_STRING;
+                        String value1 = EMPTY_STRING;
```

.. snippet ..

```
  case BrickDefine.PLAY_SOUND:
          serializer.startTag("", "command");
          serializer.attribute("", "id", Integer.toString(BrickDefine.PLAY_SOUND));
          serializer.startTag("", "sound");
          serializer.attribute("", "path", brick.get(BrickDefine.BRICK_VALUE));
          serializer.endTag("", "sound");
          serializer.endTag("", "command");
          serializer.startTag(EMPTY_STRING, COMMAND);
          serializer.attribute(EMPTY_STRING, ID, Integer.toString(BrickDefine.PLAY_SOUND));
          serializer.startTag(EMPTY_STRING, SOUND);
          serializer.attribute(EMPTY_STRING, PATH, brick.get(BrickDefine.BRICK_VALUE));
          serializer.endTag(EMPTY_STRING, SOUND);
          serializer.endTag(EMPTY_STRING, COMMAND);
```

Added some 'static finals' to make the code more maintainable!
During a re-factoring session source code was changed.
The green lines replaced the red ones.
This change makes the class more maintainable than before.
Hardcoded Strings are very bad when software needs to be changed!

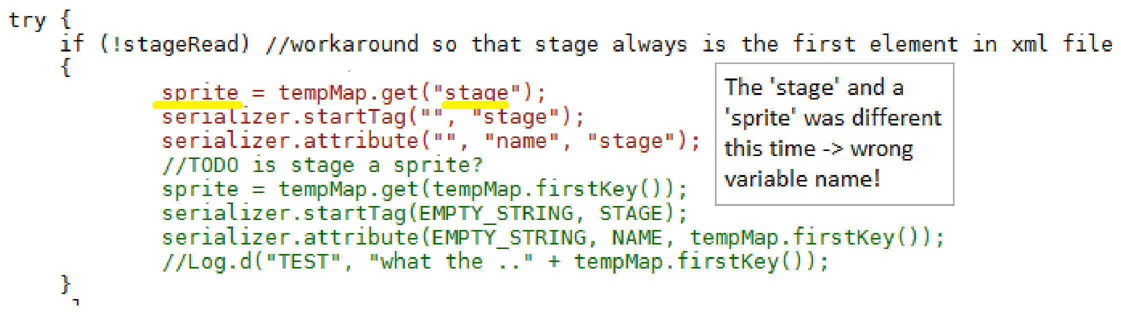
Fig. 6: Fixed problems that can make maintenance difficult.

As we can see in Fig. 6, it would take a lot of time to change that particular code. Hard-coded strings are distributed all over the parser-class. The instance of this class is obviously intended to parse an XML-based file. When the structure of that XML file needs to be altered, regardless of whether changing functionality or correcting a mistake, it could be necessary to change many of these hard-coded strings all over the parser-class.

Mistakes can easily happen when changing such a piece of code. When, for example, changing an XML tag like the id-tag, it is necessary to change it at every single place within this class. When this piece of code is not covered by test cases, a fault might be created, which would lead to further problems.

When parsing XML data many different tags and values can be combined in a varying order. This makes it almost impossible to test every likely input. As already mentioned in the chapter on quality aspects, source code that handles input data has a high fault potential, therefore this part of code should necessarily fulfil the quality aspects concerning testing and maintainability to prevent likely failures.

As we can see in Fig. 7 names of objects are misleading. This is a high potential source for creating new bugs.

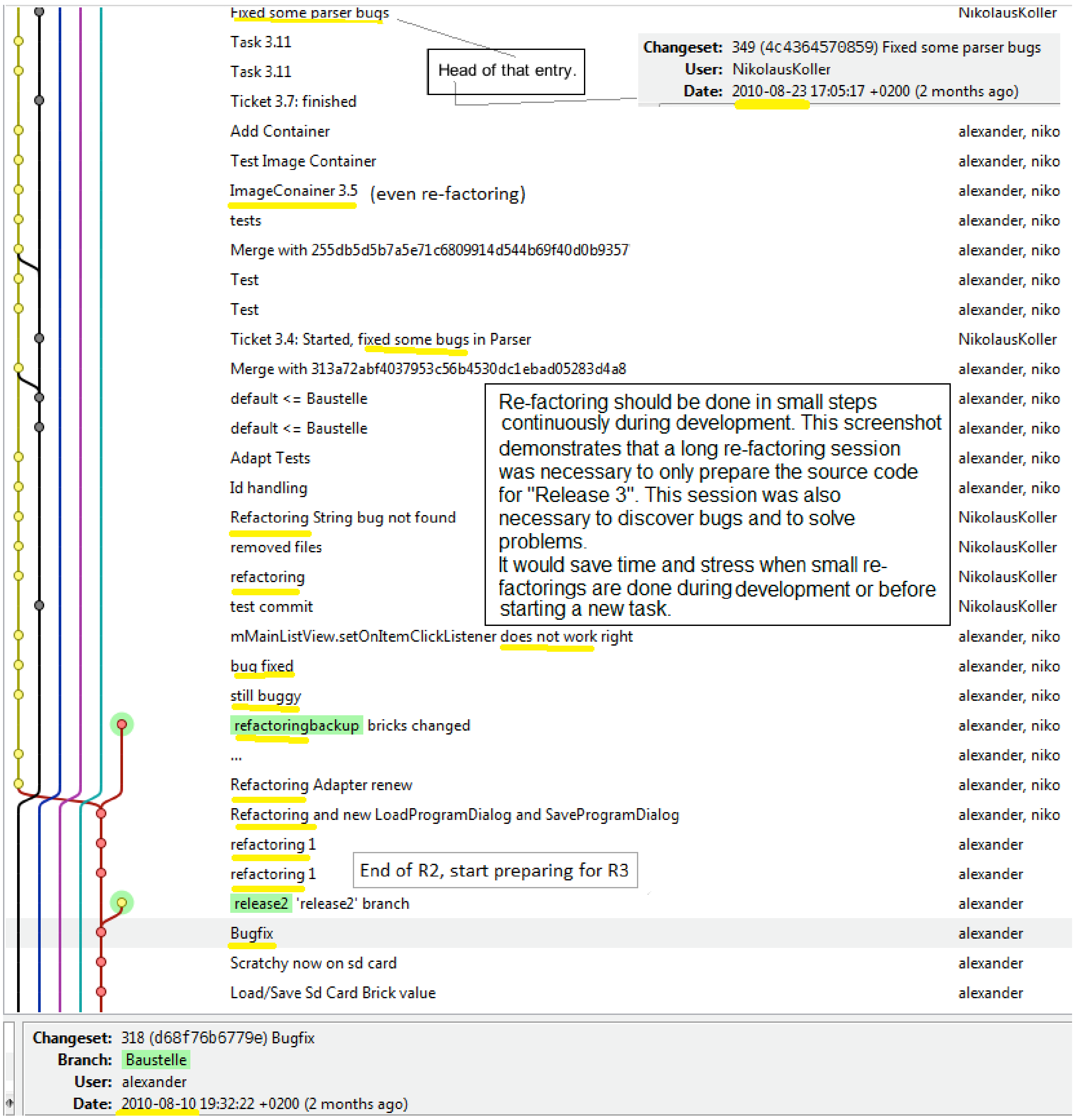


```
try {
    if (!stageRead) //workaround so that stage always is the first element in xml file
    {
            sprite = tempMap.get("stage");
            serializer.startTag("", "stage");
            serializer.attribute("", "name", "stage");
            //TODO is stage a sprite?
            sprite = tempMap.get(tempMap.firstKey());
            serializer.startTag(EMPTY_STRING, STAGE);
            serializer.attribute(EMPTY_STRING, NAME, tempMap.firstKey());
            //Log.d("TEST", "what the .." + tempMap.firstKey());
    }
```

The 'stage' and a 'sprite' was different this time -> wrong variable name!

Fig. 7: Snapshot of Mercurial, changed misleading text.

At the end of 'release2' (Mercurial tag in Fig. 8) a big refactoring period was necessary to restructure the application's architecture. The Mercurial comments shown in Fig. 8 give an overview of that period. Words like refactoring and bug fix are common.



Fixed some parser bugs	NikolausKoller
Task 3.11	
Task 3.11	
Ticket 3.7: finished	
Add Container	alexander, niko
Test Image Container	alexander, niko
ImageConainer 3.5	alexander, niko
tests	alexander, niko
Merge with 255db5d5b7a5e71c6809914d544b69f40d0b9357	alexander, niko
Test	alexander, niko
Test	alexander, niko
Ticket 3.4: Started, fixed some bugs in Parser	NikolausKoller
Merge with 313a72abf4037953c56b4530dc1ebad05283d4a8	alexander, niko
default <= Baustelle	alexander, niko
default <= Baustelle	alexander, niko
Adapt Tests	alexander, niko
Id handling	alexander, niko
Refactoring String bug not found	NikolausKoller
removed files	NikolausKoller
refactoring	NikolausKoller
test commit	NikolausKoller
mMainListView.setOnItemClickListener does not work right	alexander, niko
bug fixed	alexander, niko
still buggy	alexander, niko
refactoringbackup bricks changed	alexander, niko
...	alexander, niko
Refactoring Adapter renew	alexander, niko
Refactoring and new LoadProgramDialog and SaveProgramDialog	alexander, niko
refactoring 1	alexander
refactoring 1	alexander
release2 'release2' branch	alexander
Bugfix	alexander
Scratchy now on sd card	alexander
Load/Save Sd Card Brick value	alexander

Changeset: 349 (4c4364570859) Fixed some parser bugs
User: NikolausKoller
Date: 2010-08-23 17:05:17 +0200 (2 months ago)

Head of that entry.

(even re-factoring)

Re-factoring should be done in small steps continuously during development. This screenshot demonstrates that a long re-factoring session was necessary to only prepare the source code for "Release 3". This session was also necessary to discover bugs and to solve problems.
It would save time and stress when small re-factorings are done during development or before starting a new task.

End of R2, start preparing for R3

Changeset: 318 (d68f76b6779e) Bugfix
Branch: Baustelle
User: alexander
Date: 2010-08-10 19:32:22 +0200 (2 months ago)

Fig. 8: Mercurial snapshot that shows how long the necessary refactoring session took.

The big re-structuring at the end of the period indicates that the software's structure has been considered too little. When it is obvious that implementing new functionality would require many changes to the given source code, one should consider a refactoring and a change of architectural structure earlier. Otherwise problems and difficulties to integrate new functionality would increase.

The unavoidable refactoring work would only be delayed.

Changes for the period starting on 2010-09-16

The developer needs to analyse the task more carefully and plan them to fit into the system. At first it is important to understand the given code that the new task should be based on. When the developer does not know or understand the software's architecture he can easily lose the overview, and design faults are likely. Furthermore he has to decide if refactoring the source code is necessary. The focus has to be kept on testing and in achieving high code coverage.

When the given source code is tested and understandable the task should be analysed carefully, because a well explored situation may help to find meaningful names and descriptions. To give special focus to this topic a new column named “Analysis” is introduced in the Kanban board. This should encourage the developers to take this analysis part more seriously.

The deploy lane description is extended with “Test” and “Code review”. This adjustment should remind the developer to integrate only tested source code and that another developer should be asked to review the code. This should force communication between developers and keep them up to date.

Bugs make it difficult to evaluate the actual work in progress and the put-through rate. Special bug tickets on the Kanban board should make a slow progress obvious. They will differ in colour. Many bug tickets on the board indicates that something is going wrong and that there is no or hardly any progress.

The new Kanban board is shown in Fig. 9.

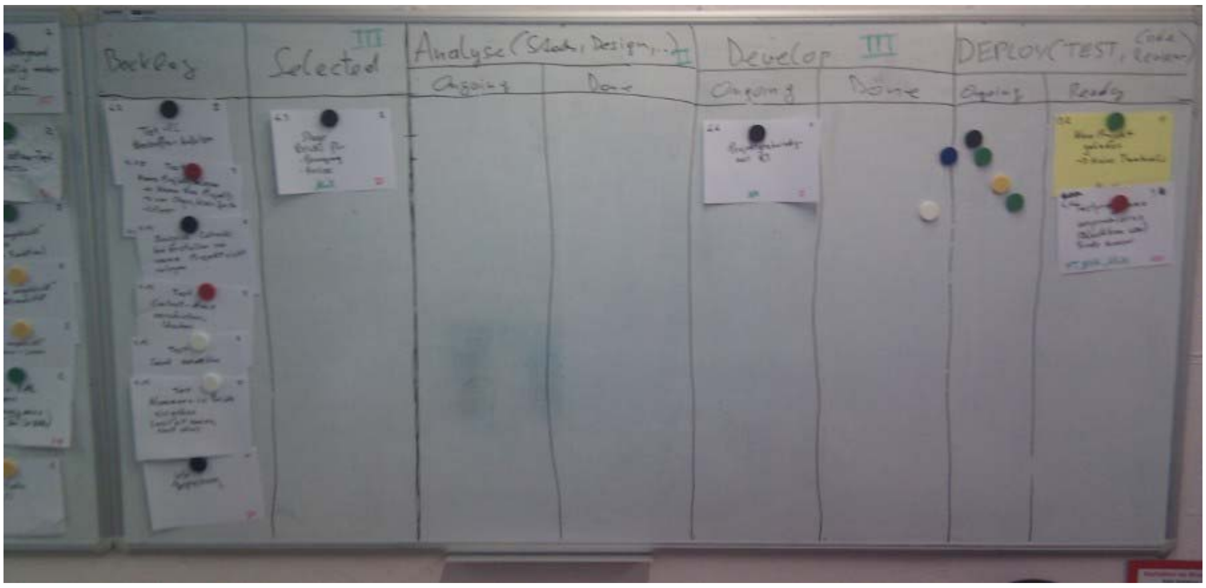
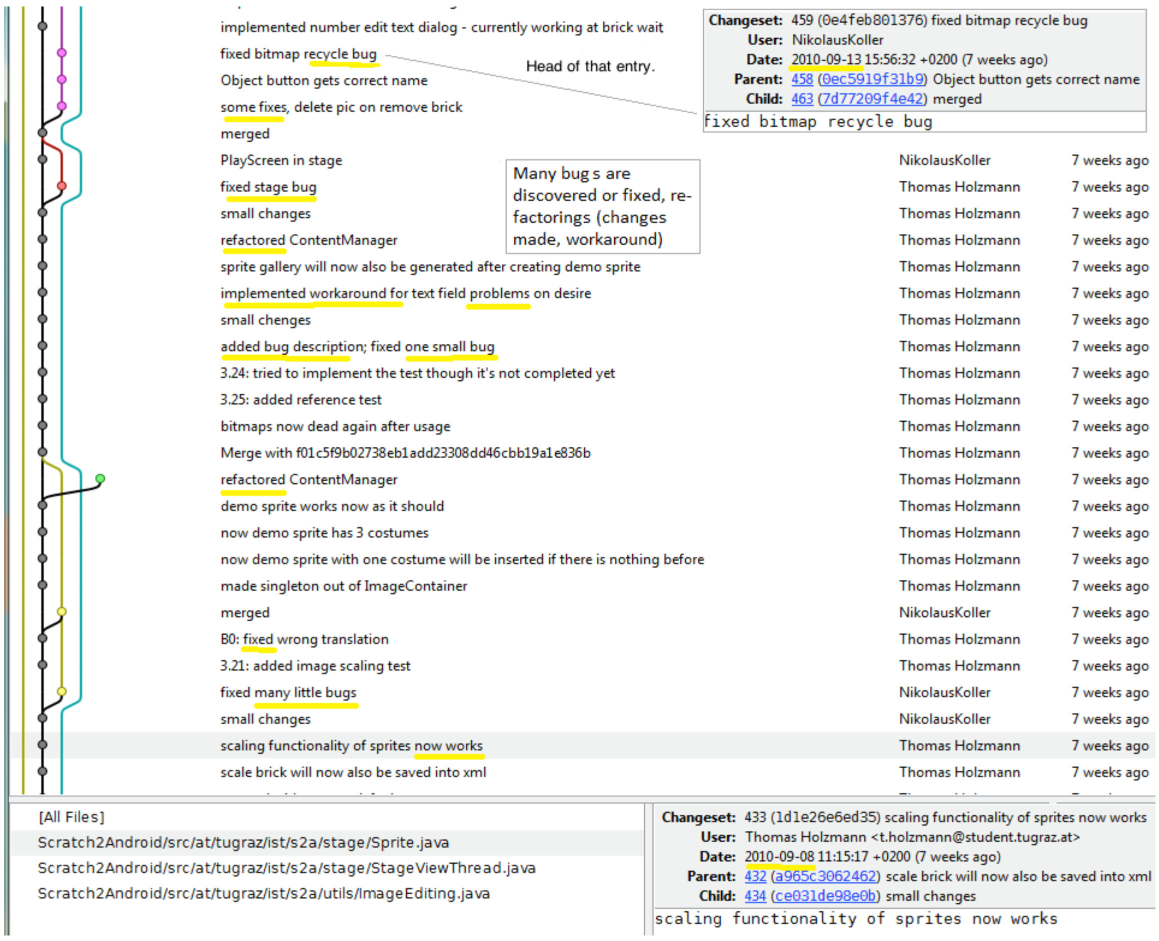


Fig. 9: Picture of the Kanban board from the period starting on 2010-09-16

4.3.4. Adjustment on 2010-10-04

The most noticeable problems are similar to those in the last period. The last changes seem to have made no obvious difference. More bugs and many problems need special attention as shown in Fig. 10.


```
implemented number edit text dialog - currently working at brick wait
fixed bitmap recycle bug
Object button gets correct name
some fixes, delete pic on remove brick
merged
PlayScreen in stage                                                      NikolausKoller     7 weeks ago
fixed stage bug                                                          Thomas Holzmann    7 weeks ago
small changes                                                            Thomas Holzmann    7 weeks ago
refactored ContentManager                                                Thomas Holzmann    7 weeks ago
sprite gallery will now also be generated after creating demo sprite     Thomas Holzmann    7 weeks ago
implemented workaround for text field problems on desire                 Thomas Holzmann    7 weeks ago
small chenges                                                            Thomas Holzmann    7 weeks ago
added bug description; fixed one small bug                               Thomas Holzmann    7 weeks ago
3.24: tried to implement the test though it's not completed yet          Thomas Holzmann    7 weeks ago
3.25: added reference test                                               Thomas Holzmann    7 weeks ago
bitmaps now dead again after usage                                       Thomas Holzmann    7 weeks ago
Merge with f01c5f9b02738eb1add23308dd46cbb19a1e836b                      Thomas Holzmann    7 weeks ago
refactored ContentManager                                                Thomas Holzmann    7 weeks ago
demo sprite works now as it should                                       Thomas Holzmann    7 weeks ago
now demo sprite has 3 costumes                                           Thomas Holzmann    7 weeks ago
now demo sprite with one costume will be inserted if there is nothing before   Thomas Holzmann    7 weeks ago
made singleton out of ImageContainer                                     Thomas Holzmann    7 weeks ago
merged                                                                   NikolausKoller     7 weeks ago
B0: fixed wrong translation                                              Thomas Holzmann    7 weeks ago
3.21: added image scaling test                                           Thomas Holzmann    7 weeks ago
fixed many little bugs                                                   NikolausKoller     7 weeks ago
small changes                                                            NikolausKoller     7 weeks ago
scaling functionality of sprites now works                               Thomas Holzmann    7 weeks ago
scale brick will now also be saved into xml                              Thomas Holzmann    7 weeks ago
```

```
Changeset: 459 (0e4feb801376) fixed bitmap recycle bug
     User: NikolausKoller
     Date: 2010-09-13 15:56:32 +0200 (7 weeks ago)
   Parent: 458 (0ec5919f31b9) Object button gets correct name
    Child: 463 (7d77209f4e42) merged
fixed bitmap recycle bug
```

```
[All Files]
Scratch2Android/src/at/tugraz/ist/s2a/stage/Sprite.java
Scratch2Android/src/at/tugraz/ist/s2a/stage/StageViewThread.java
Scratch2Android/src/at/tugraz/ist/s2a/utils/ImageEditing.java
```

```
Changeset: 433 (1d1e26e6ed35) scaling functionality of sprites now works
     User: Thomas Holzmann <t.holzmann@student.tugraz.at>
     Date: 2010-09-08 11:15:17 +0200 (7 weeks ago)
   Parent: 432 (a965c3062462) scale brick will now also be saved into xml
    Child: 434 (ce031de98e0b) small changes
scaling functionality of sprites now works
```

Head of that entry.
Many bug s are discovered or fixed, re-factorings (changes made, workaround)

Fig. 10: Mercurial screen shot that illustrates the number of problems in this period.

At the end of the period there are too many problems that need the developer's attention. These problems should have been solved before or during development and not kept until the end.

The put-through rate is hard to evaluate and the application's functionality seems to be stuck, which can be shown by Mercurial entries.

The feedback of some developers points out interesting information concerning the given problems. Some of them were talking about design faults that make it hard to extend functionality. This can even be proved by the Mercurial entries. Some of them struggle with the platform and the testing possibilities. This might be the reason for the absence of test cases.

Tickets were observed to jump between the lanes. This can mean that the “analyse lane” and the “deploy lane” are being ignored, but more likely indicate that the Kanban board in this form is not an effective representation of the actual development process. There could also simply be a problem of understanding the meaning of those lanes. There might even be a problem with the structure of the Kanban board. When a task cannot be finished for some reason, where to put the ticket would be the question. This might be the reason for finishing a ticket (move it in the done section of the last lane) without actually finishing the task.

Changes for the period starting on 2010-10-04

The next consequential step to prevent bugs is writing test cases. When the developers try to define test cases before starting with the actual programming of the task, they have to concern themselves with the difficulties of testing on Android without being distracted by developing functionality. So the “Analyse” lane is re-named into “Test First”, which should increase the number of test cases and should decrease the number of faults. To be able to write test cases it is necessary to analyse and design the software, therefore this new lane requires code analysis. An obvious problem with this approach is that in test driven development, the tasks of writing tests and writing normal code are constantly alternated within one pairing session, so a consequent use of both the test-first and the development lanes would mean that one has to move cards frequently back and forth between these lanes.

The team is constantly growing. A new subproject has been started. Therefore the board is horizontally split into two parts. The second part should give space for the new subproject that needs to be visualized separately. Developers should have the possibility to perform tasks on both project parts to spread know how.

The new Kanban board is shown in Fig. 11.

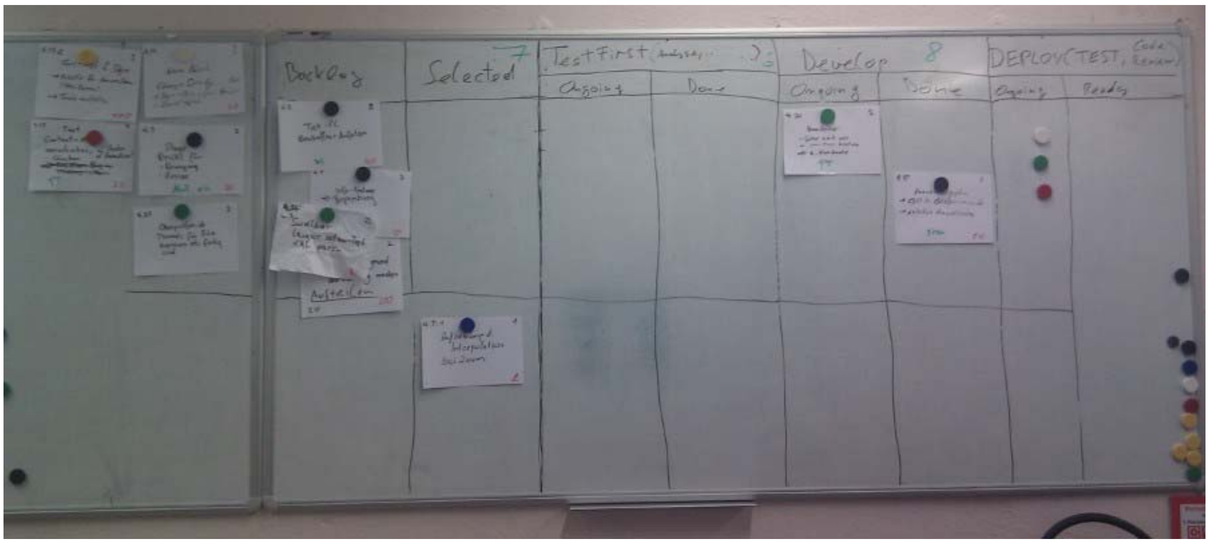


Fig. 11: Picture of the Kanban board from the period starting at 2010-10-04

4.3.5. Adjustment on 2010-11-26

The most noticeable problems are still many bugs, which is shown by the collected bug tickets and by the entries commented in Mercurial. This means that there is still low reliability [CatMerc10]. One could wonder whether this number would be higher or lower when using a different development process given the other circumstances of the project, such as fairly complex requirements, a large number of developers with limited previous skills in the domain and employed technology that work only part time and also at different times.

The number of bug tickets is increasing constantly. It seems that the more the application's functionality grows the more the number of faults increases, which also was to be expected and is normal for a project of this scope. Functionality that was once working correctly does not work anymore, since bug descriptions have appeared that have already been solved before. All this obviously leads to a lower put through rate.

Some of the experienced developers call for a prolonged refactoring period to clean up the code and build a new basis for the next extensions of the project.

Changes for the period starting on 2010-11-26

Introducing more communication and control is the step that will be applied next. The experienced developers know about the software's architecture, since they built it. They gained skills in writing tests on this platform. Therefore they will get the responsibility to control integrations of software features and adjustments in a pair-session with the developer who is assigned to that task. They will also provide time slots in a week to support the less experienced developers in becoming an advanced one. These members are called core-members.

The Kanban board is reduced to a minimum. Now there is just a backlog lane, a development lane and a done lane. There is no ticket limitation anymore, since a developer can only be assigned to one ticket. There is no way to stop or finish that task without having a discussion with one of the core-members.

The new Kanban board is shown in Fig. 12.

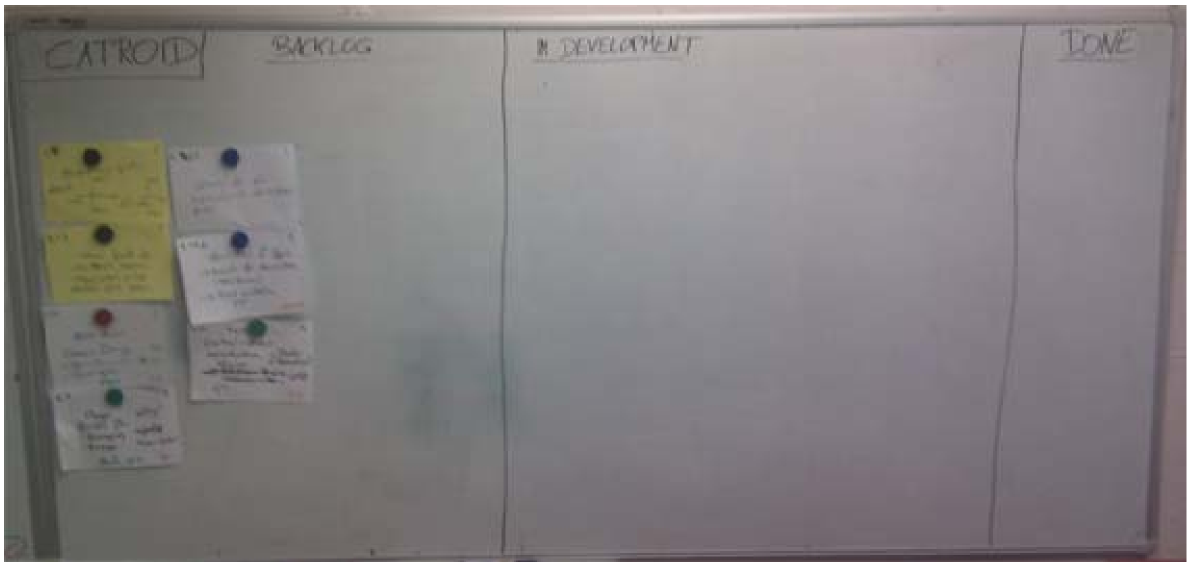


Fig. 12: Picture of the Kanban board from the period starting on 2010-11-26

4.4. *Monitoring summary*

On the whole the team seemed to be a little overextended, which resulted in neglecting their duty to achieve high testing code coverage. They have seemed to be more concerned with the new technology - the mobile platform.

A desired development progress

The following diagram in Fig. 13 shows an example of a development process path that is possible if the focus is constantly kept on quality.

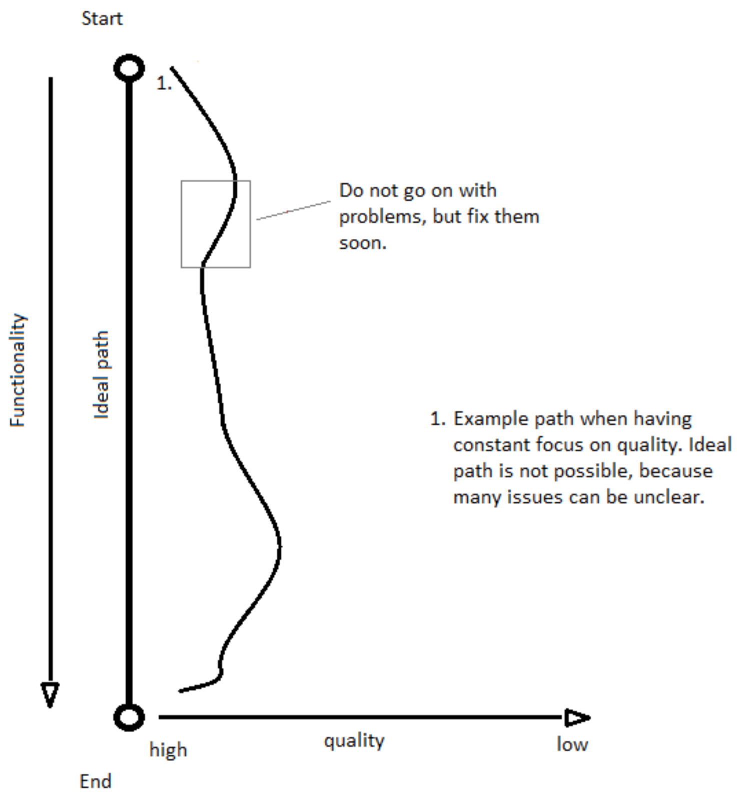

Start
1.
Do not go on with problems, but fix them soon.
Functionality
Ideal path
1. Example path when having constant focus on quality. Ideal path is not possible, because many issues can be unclear.
high
quality
low
End

Fig. 13: Software development period. Can either start from scratch or be based on given code.

As we can see in Fig. 13, we have got an example path of a development progress that shows the software's quality at a given point. No project can be expected to run the ideal path. There are many factors like a new platform, new techniques or an unclear goal that are daily life in this business and additionally the value of quality is subjective. This example path simply demonstrates how a development process could go when keeping the focus on quality. This can even mean, accepting an extension to development time when necessary to keep the code at a high level of quality.

An expansion of development time can be positive in so far as it is used to clean up code. Furthermore it seems to be normal since no source code will ever be a hundred percent perfect. Such an effect can for example happen when a developer encounters low quality code and refactors it. The time it takes to finish the actual task increases by an additional analysing and refactoring period. Potential faults can therefore be avoided and new functionality can be based on checked code. The project's lead time can still be estimated in a realistic way, since it should not be influenced by undiscovered faults, at least by no serious ones.

The black path in Fig. 13 demonstrated such a possible situation. The little curves the path takes, illustrate a short expansion of development time. It slows down the progress of functionality but keeps the project near the ideal path of quality. In such cases, encountering lower quality code has a short-term effect, if a refactoring is performed.

The development progress mainly detected between 2010-09-16 and 2010-11-26

Fig. 14 demonstrates an abstracted path of quality similar to the observed one, compared with a desired example path.

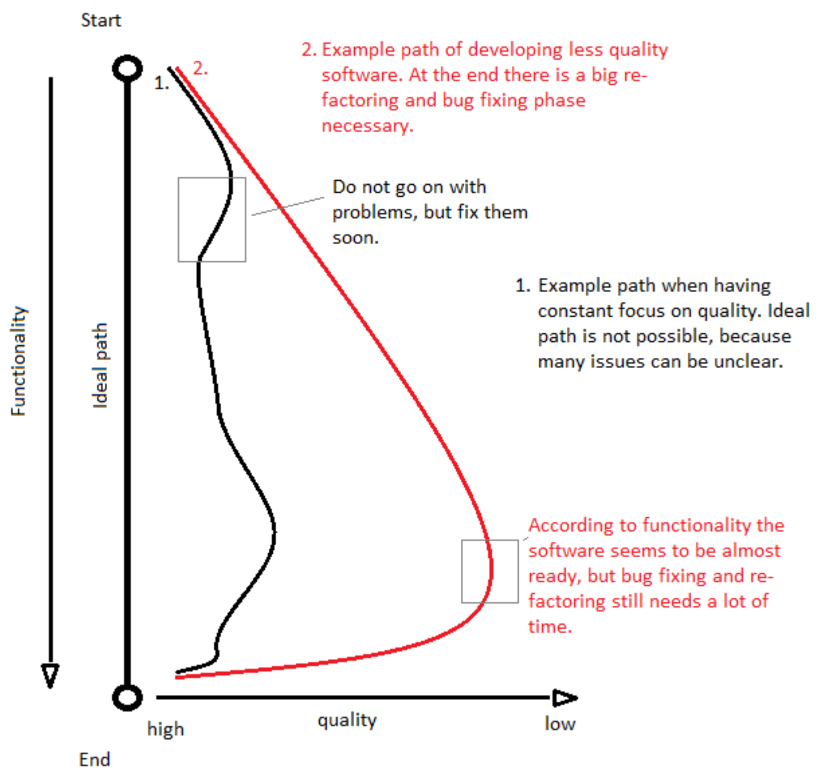

Start
2. Example path of developing less quality software. At the end there is a big re-factoring and bug fixing phase necessary.
1.
2.
Do not go on with problems, but fix them soon.
1. Example path when having constant focus on quality. Ideal path is not possible, because many issues can be unclear.
Functionality
Ideal path
According to functionality the software seems to be almost ready, but bug fixing and re-factoring still needs a lot of time.
high
quality
low
End

Fig. 14: The red path shows what can happen when relying on lower quality code and going on with that quality basis.

What obviously happened is shown by the red path in Fig. 14. New functionality is based on lower quality code. This situation could have resulted for the following reasons:

A developer encounters low quality code with non-satisfying test cases and no refactoring is performed. The source code is not even analysed accurately or verified for correctness. In this situation, there might be software faults that will lead to failures. The developer tries to implement the new task on the current low quality version and thus undiscovered faults can be dragged through the project and further problems are likely.

When this is done again and again the situation can dramatically worsen. Even small changes can now harm the software. Much functionality may have to be re-implemented and major refactoring periods need to be performed. A developer might be demotivated and stop feeling responsible, which could result in a vicious circle.

Ignoring low quality code can have a long-term effect on the software development process. The remaining amount of work can hardly be estimated anymore. Furthermore, it can be difficult to find out what functionality works correctly.

Indications for an expanding development time

When starting a task several paths can be taken which will have a different effect on the project's progress. Each of the chosen paths can have a different influence on the quality of software as has been illustrated before. Two situations can be an indication for a long- and short-term effect. A closer look would be needed to classify the situation.

- The developer talks about problems during a stand-up or a private meeting. In this case, the developer gives concrete information about problems he is concerned with.
- The note of a repository commit (e.g., Mercurial) can give information about the particular situation. Keywords like "refactoring", "bug fixing" or "problems" give crucial information, but even "start task X" or "finish task X" are helpful in analysing the situation.

There are different signs that can indicate positive short-term effects when monitoring progress and help to understand the actual situation. One of them is the following:

- A Kanban ticket becomes stuck in a lane. The developer might struggle with quality problems. This can only be determined by having a closer look. The Kanban system provides an easy possibility to follow the project's progress by limiting tasks in progress.

A negative long-term effect on the software development process can be the result when low quality code is ignored. The following facts can be indications for a long-term effect:

- An increasing number of bugs.
- A decreasing put through rate. This happens when many tickets are stuck. Many appearing failures might decrease the put through rate, since no new task can be started when bugs need attention.

All those indications might also have other reasons for occurring, so a review is necessary.

4.5. A simple way to discover quality faults

Monitoring the progress of development can be the first step for discovering problems without having any code review. An unexpected development time may be an indication of having low quality code. In the section above (Summary) there was a positive short-term effect and a negative long-term effect mentioned. Observing these effects could help to discover quality faults during the development process and the project would still stay lean and agile.

Note that this method alone would not replace a detailed software evaluation but can be used in addition to the indications found and mentioned above to alert developers when the software's quality path drifts too far from the desired value.

Influence on task-finish times

Many factors play an important role when determining how long it takes a developer to finish a task. Simple crucial factors are the power and the motivation of a person at a particular development time. In this case a developer is characterised by his awareness of quality and experience of using a particular platform. These values seem to be the most crucial ones as has been observed in this project

The following table illustrates a raster that helps to estimate the time it takes a developer to finish a task relatively to the average expectation.

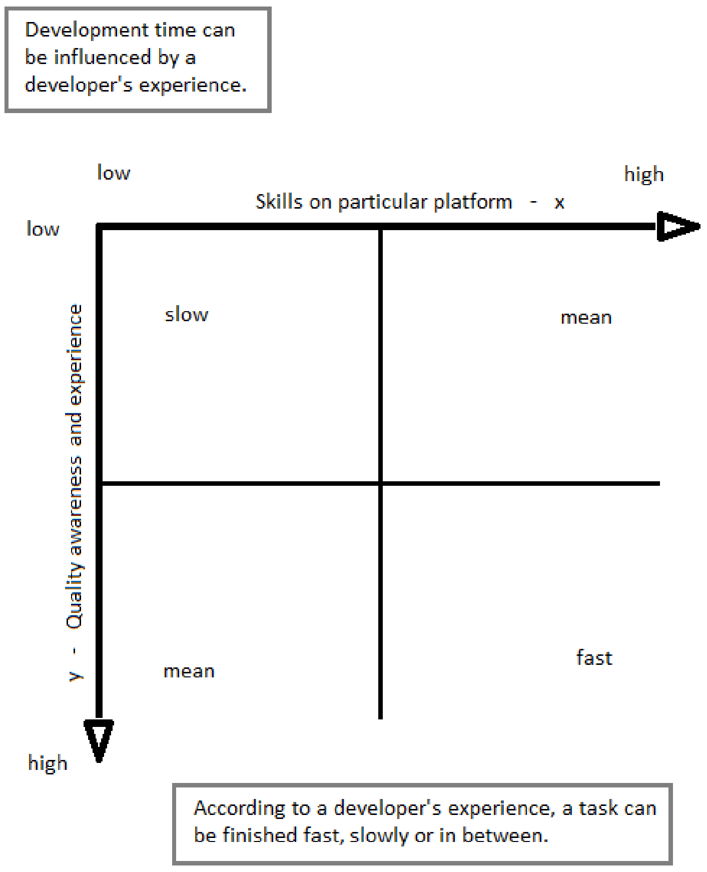

Development time can be influenced by a developer's experience.
low
high
Skills on particular platform - x
low
slow
mean
y - Quality awareness and experience
mean
fast
high
According to a developer's experience, a task can be finished fast, slowly or in between.

Fig. 15: A simplified raster that helps to estimate a developer's task-finish-time factor based on his experience.

In Fig. 15 a simple raster is shown that classifies a developer's programming speed according to his software development experience. It provides a simplified method to determine a value that has influence on a task-finish time. Two factors are considered as one of them is based on the experience a developer has within a particular platform, like Android. This is illustrated on the x-axis. The awareness of quality and the skills a developer has are shown on the y-axis, which constitutes the second factor in this table.

It is assumed that a quality and platform experienced developer needs less time to write quality software than a less skilled one. Even when a developer is highly skilled in writing quality code it is necessary to explore a platform first to find a way for achieving high quality code.

We will move now to consider how the table in Fig. 15 can be beneficial when evaluating the quality of code by the task-finish time. The determined table value can help to estimate a development time needed to finish a task. A variation from the actual taken time to the expected one can be an indication of problems, but does not have to be one, since more factors play a crucial role as already mentioned before. Nevertheless it is worth having a detailed look at it when a variation of a task-finish time is noticed, as one can see in the next section.

Generally speaking, when starting a task there are two different situations a developer can be confronted with. In the first situation there can either be a clean and tested given code, fulfilling most of the quality aspects, or there can be a second situation with source code that does not or hardly satisfies any quality requirements. Starting a task without having any fundamental code would be classified into the first situation. A rough indication of having the first or the second situation can be given by values like the test case coverage of source code and other stated quality requirements respectively. It is of course not possible to ever classify a situation precisely, however this would not necessarily be desired.

The first situation would be normal when a task takes about the estimated time relative to the determined table value. When a task-finish time varies significantly within the first situation, one of the three following reasons might be responsible:

- The development time was wrongly estimated.
- The developer has been overwhelmed by the task, because of his lack of experience.
- The task has not been done in a way that ensures quality, since it takes time finding test cases, writing readable code and considering software's design that can be easily integrated into the given structure.

The third reason can lead to the second situation a developer can be confronted with, when the given code does not or hardly fulfils the quality criterion.

When being confronted with the second situation the normal development time would be the expected one, the same as in the first situation, plus some extra time that is necessary to analyse and refactor the given code. If the code is less readable or has an unclear structure, it is hard to understand and therefore requires a detailed analysis. When no test cases are available that verify functionality, it might not be clear if the code is working correctly. To ensure basing new functionality on a correct computing code the engineer needs time to test. This is one of the reasons why quality faults extend the development time, which would be a productive step, because a potential mistake would not affect further functionality.

4.6. The final situation

After analysing the information gained within this chapter, new rules can be created. Problems that might occur are defined and rules that would prevent them can be introduced. The information found can be used to understand how agile techniques can be applied and what for. It has been discovered how developers might deal with new methods and techniques in this particular situation. According to this data a recommendation can be made that supports integrating new developers into the team.

The applied adjustment

Some of the rules that were created during the monitoring phase can be adopted, but some adjustments are necessary. These are summarised by the following text:

The Kanban board looks as shown in Fig. 12 above. Now there are three kinds of tickets instead of two.

1. The “Story card” is extended by the name of the card creator, by the name of the developer who implements it and by the name of the person who accepts it to be finished. This should force a controlled task managing.

2. The “Usability card” is new and is additional to the “Story card”. There can be additional text and a mock-up on it to describe tasks more precisely.

3. The “Bug card” stays the same as before. The main difference to a “Story card” is its difference in colour.

The main change, which seems to be very effective, is the fact that a task needs to be accepted before it can be integrated into the system. Another team member (any other project member will do as this will allow not to slow down the development process and also is a good way to spread the knowledge about the newly created code; if the 3rd member is from outside the subteam, it has the additional benefit that the inspection is done by someone with a fresh view on the code and thus the code inspection can be done in a more unbiased and critical way) has to accept a ticket, checking adherence to the following points:

1. The source code has to be clean and readable (no abbreviations are allowed, no misleading names, the only language used is English) as it constitutes a major part of the documentation for other developers and thus needs to be easily understandable by 3rd persons like the person who does this acceptance

2. Comments should only be written when absolutely necessary. The description of functionality is done by tests.

3. The rules of test-driven development were followed. Their completeness can actually not be made sure quickly by the 3rd person during this walk-through, but the new tests should be at least looked through superficially and their completeness at least discussed with the pair that wrote them. Particular consideration should be accorded to their readability as they constitute the 2nd main part of the documentation, the 1st part being the main code itself.

4. The architectural design needs to be considered (no code duplications, design patterns should have been applied as reasonable).

5. If the acceptance fails, the developers have to correct it and then **resubmit** it to this or another new acceptor. the story card thus remaining in the development lane, or, if the requested changes are too big, they have to write new stories together with at least one other person which is ideally the person who did the acceptance that failed, though it also can be any other team member in order not to slow down the development process)

6. Only when all the checks satisfied the acceptor, is the source code integrated into the main branch of the Mercurial repository, after which its story card is moved to the done & accepted lane on the Kanban board.

The result of the survey

An anonymous survey was done after applying the rules mentioned above for two months, at the end of February 2011. The developers that worked in January and February were questioned absolutely anonymously concerning the project. They were asked about the main topics that have been discussed in this thesis. The result of the survey is summarised by the following points:

- The developers are satisfied with the team atmosphere. Everybody feels free to speak his or her mind and to make suggestions for improvements. The team members are keen to help each other.

- The Kanban system is accepted and appreciated in its simplified form (as shown in Fig. 12). Nevertheless, the tasks are often criticised as being too dependent on other tickets or knowledge. However, this latter aspect is normal for any project and cannot be avoided.

- The beneficial values of test-driven development are accepted, but there is not always interest in following its rules accurately, especially when the developers are confronted with difficult testing situations.

- The source code is evaluated as readable and mostly bug free from the time on the code-acceptance process was introduced.

- Almost all the developers think that testing has high value, but there has to be improvement in testing quality, thus they call for a detailed testing guideline to prevent less meaningful tests and to accelerate finding test cases within particular situations.

- Tests as documentation have been less accepted. Many of the developers want to have a combination of tests and traditional documentation such as comments and Java-doc. One of the reasons could be that not all developers had any theoretical introduction to test-driven development at all and thus do not clearly understand that the test of the test-driven development process serve less as tests of traditional development process and much more as 100% correct documentation allowing to extend code of other developers without having to study textual documentation whose correctness is always questionable and in real life situations is often much more incomplete than the documentation-type tests written in test-driven development. As a consequence, the education of team members w.r.t. test-driven development should be done in a more thorough way before they can start developing.

- Further suggestions are: A thought-out architectural design that can be kept for future versions and a clear direction for the project. Again, this is most likely based on a misconception regarding the principles of test-driven development or rather, the YAGNI[22] principle of the extreme programming agile development method that is most likely unknown to most team members. As a consequence, the method should be introduced and made clear to the developers.

Progress can be recognised now and the team seems to feel comfortable in this new situation. Some small improvements need to be made but the overall system seems to be successful.

5. Conclusion

In this chapter the applied techniques are evaluated along with how the developers handled them.

Once again the quality aspects that were considered in this document are reliability, time performance, space performance, portability and maintainability.

Time and space-performance were hardly influenced by the techniques applied. Both topics are affected by the architectural design and thus could be evaluated during code-review. Knowledge in writing fast algorithms[23] would be the most important issue concerning these two topics. Neither Kanban nor test-driven development have any influence on them. The focus of this work is mainly set on the other topics mentioned. The following sections illustrate how they were affected.

5.1. Testing

Testing seems to be really important in a dynamic team that constantly changes its members. When correctly applied, testing can reduce stress as a developer feels safe when maintaining or extending software. Furthermore test cases can guide a developer through his work as documentation and task definition. Nevertheless it is not that simple to achieve satisfying results. Several preconditions have

[22] YAGNI: “you aren’t gonna need it” (see “Extreme programming explaned: Embrace Change” by Kent Beck, Addison Wesley, 2001).
[23] Writing fast algorithm: keyword: O-Notation

to be fulfilled.

Tests are also fundamental to ensuring that all developers can work on any part of the code without having to ask other developers for further details: all details are given in the code and the test cases. This gives further motivation to developers to document their code with tests and clear code.

Monitored precondition for a successful testing

Writing test cases and achieving high code coverage can be difficult without experience in this field. It might be helpful to introduce a developer to the different testing methods and possibilities to ensure a beneficial result. Otherwise general testing difficulties given by the object-oriented paradigm, input/output issues or unexplored system features can prevent success.

Another barrier can be design problems that make it difficult to write test cases. Furthermore they are a potential source of new faults. The developer might struggle when writing simple tests. A platform needs to be analysed at first to be able to make beneficial designs. A faulty design makes not only implementing new functionality difficult but even writing simple tests, which can lead to an unsatisfying test coverage. A less maintainable and reliable software would be the consequential result. A sophisticated architectural design is essential for high code coverage and therefore gaining skills on the platform is an important precondition that needs to be fulfilled.

Furthermore, mobile applications on Android use many system features which are the most difficult part to test on this platform and therefore require special attention. Many useful APIs provide the advantage of a fast implementation but do lead to a loss of control. The fact that system features are often tightly connected to one’s own code makes testing difficult.

Variations in the results of hardware and GUI communications do not only lower the value of portability, but even make fully automated testing hardly realisable. Functional tests would help when testing particular features, but they cannot test issues like vibration of the mobile phone or flashing LED[24] lights. Therefore manual testing would be necessary to ensure a high testing coverage.

[24] LED- Light Emitting Diode

A disadvantage is that nothing makes sure these manual tests are actually performed. Thus, a semi-automatic testing method might help, where tests are integrated into the automatic tests but need user participation to check results which are then communicated by the user to the test engine and checked there against expected results.

5.2. The team

A free and agile system gives each team member the possibility to ignore rules when he is not able to follow them (e.g., due to low skills) or if there is little concern about the application due to loss of interest. It is necessary to mention that the rules created during the development process were not defined precisely but by experimental ideas. They were meant to see if small variations in the applied techniques can have effect on the project's quality. When particular rules make no sense from a developer's point of view or cannot be understood, they might obviously be ignored.

Nevertheless, within this special team situation it seems to be important to ensure an information and knowledge infrastructure, to keep the team up to date and give them a chance to get questions answered. A new team member has to deal with a platform and agile methods that are possibly new to him. This might be the reason why there were so many quality problems all through the development process. A beginner can hardly apply all of the rules. Therefore it would be important to introduce them more precisely to the desired techniques and rules.

So it is important to assign responsibility to experienced members and give them control. Control would be the important factor to balance the given responsibility, since somebody should not be held responsible for something he has no influence on. They need to delegate the project's tasks and developers. Otherwise nobody would feel responsible for problems.

The team atmosphere and the individual developer play a big role in creating high quality software. Skilled and motivated people are essential for achieving high quality code as well as for being successful when applying agile and lean methods.

5.3. Kanban

The Kanban system is a simple way to administrate tasks. It is easy to understand and gives an overview about the work in progress.

The number of tickets and their colour give an impression of the actual progress. There is a big variation of lanes and tickets possible which let one easily adjust the system to your individual needs.

The fact that a Kanban board is limited in space and that it is limited to one place could constrain a team. A sophisticated software solution could help in this case but would have the drawback to be easier to ignore and also makes it more difficult to get the whole picture and thus more difficult to gain an overview and understanding for the project itself.

The Kanban system had within the practical application no obvious influence on the quality of software. The administrative value seemed to be practical, but using the system to force particular topics like testing seems not to be valuable in the way it was used so far.

The Kanban system is considered to move the tasks in one direction. The task limitations of each lane could block the moving back of a task. When a task is rejected in a later lane, like the review lane, it is not clear where to move it to. To put it back to the lane of development could not be possible because of limitations while such limitations were still enforced. To correct the mistakes in this review lane would make it a mixture of developing and reviewing. Another way would be to move it back to the start. This would break the rules of priority in Kanban. At least there would be the possibility to make one lane for developing and reviewing. Fig. 16 illustrates that problem.

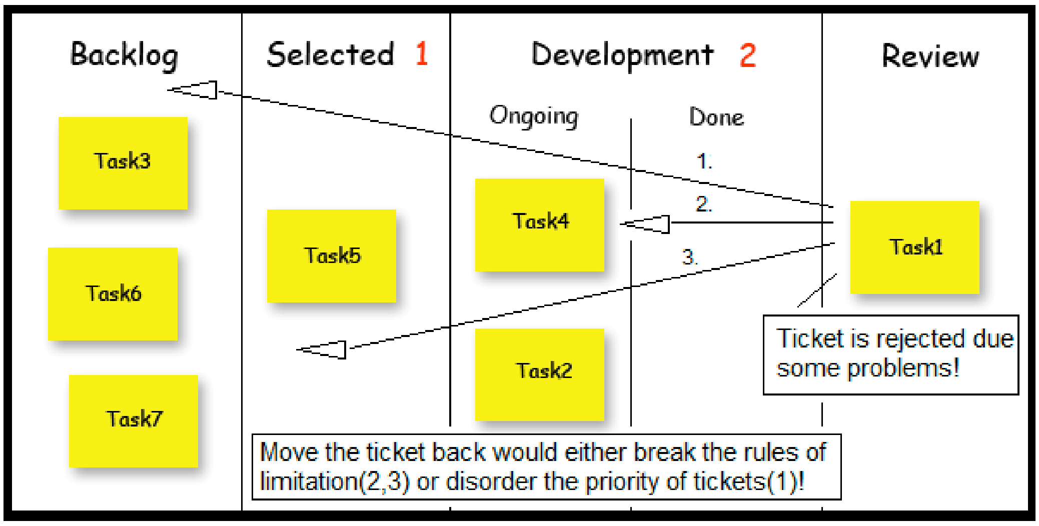

Backlog
Selected 1
Development 2
Review
Ongoing
Done
Task3
Task6
Task7
Task5
Task4
Task2
Task1
1.
2.
3.
Ticket is rejected due some problems!
Move the ticket back would either break the rules of limitation(2,3) or disorder the priority of tickets(1)!

Fig. 16: Where to move the ticket when rejected?

Despite being common in many examples, a lane for analysis introduced a lack of clarity in the process. Analysing and designing is often part of actual developing itself. Therefore analysing, designing and developing are often deployed as a circle. So it does not fit into a Kanban board, which is meant to go straight in one direction similar to a waterfall model. This is out of fashion today, precisely because it has been shown to be ineffective in almost all practical cases. When putting those lanes together the Kanban board would have hardly any value for a developer and a simple administrative board would remain.

Kanban and its crucial principles can push controlled progress. It can lower administrative work and can therefore give the developers more flexibility. The principle of finishing a task completely forced by an integration control and a limitation of tasks can have a positive effect on the value of reliability and maintainability.

5.4. Test-driven development

Test-driven development is another way of developing software. It is another way of starting and finishing a task. When starting to write some functionality there is already a test case written that will check your work. At the end there should be no need to write further test cases. The possibility of writing simple tests is the confirmation of a simple and clear software design. So writing and designing source code according to simple test cases pushes for a valuable design, which is necessary for high quality software.

There are preconditions that have to be fulfilled before applying test-driven development. First a developer has to welcome that new style of writing software. He has to understand the principles and possibilities of testing. The most important precondition is to get experience on the new platform and on how special issues can be tested. Otherwise test-driven development would not be applied successfully.

When those preconditions are fulfilled, test-driven development can boost reliability and maintainability.

5.5. Further ideas

- Testing in Android could be better supported on particular topics like the control of hardware. Some rules in changing Android (e.g.: GUI) would help to increase the ease of portability.
- User interfaces are a fundamentally important topic when creating quality software. Therefore the optimisation of their usability needs to be tightly integrated in the development process at a core level.
- Some quality aspects like security would need special attention. Especially on Android, as an open source project, this might be very interesting, because many features like SMS or telephony service can easily be used. When those features get abused, financial harm can be the consequence.
- The applied techniques tested in a team that is less dynamic would be an interesting comparison.
- How to develop a Kaizen culture needs further attention and could be an interesting addition to this document.

6. Project documents and source code

- The project of the studied team is named “Catroid” and is released under the GNU General Public Licence v3. It is under continuous development by students of Graz University of Technology. More on the open source Catroid project can be found on the Web[25] where also example applications can be found[26].
- NkNightClock which shows the varying hardware results can be downloaded here[27].

[25] Catroid open source project location (March, 2011): http://code.google.com/p/catroid/
[26] Program examples are on (March, 2011): http://www.catroid.org/
[27] NkNightClock on Android Market (March, 2011): https://market.android.com/details?id=at.nk.nightClock

Table of figures

Literature

[**AmmannOffutt08**]	Ammann, Paul & Offutt, Jeff: Introduction to Software Testing, Cambridge University Press, 2008
[**AndDev10**]	Google Inc.: Android Developers, 2010 http://developer.android.com/index.html
[**Anderson10**]	Anderson, David J.: Kanban, Blue Hole Press, 2010
[**AndMarket11**]	Google Inc.: Android Market, 2011 https://market.android.com/details?id=at.nk.nightClock
[**AndNDK10**]	Google Inc.: What is the Android NDK?, 2010 http://developer.android.com/sdk/ndk/index.html#overview
[**AndPerf10**]	Google Inc.: Designing for Performance, 2010 http://developer.android.com/guide/practices/design/performance.html
[**AndPlat10**]	Google Inc.: Platform Versions, 2010 http://developer.android.com/resources/dashboard/platform-versions.html
[**AndSystem10**]	Google Inc.: What is Android?, 2010 http://developer.android.com/guide/basics/what-is-android.html
[**Beck03**]	Beck, Kent: Test-Driven Development, Pearson Education, Inc, 2003
[**CatMerc10**]	TUGraz: The Catroid Project, 2010 https://intra.ist.tugraz.at/hg/scratch
[**Devagile10**]	Devagile: http://www.devagile.com/, 2010
[**Devices10**]	Google Inc.: Google Phone Gallery, 2010 http://www.google.com/phone/#
[**Gizmodo10**]	Brian Barrett: See Android Market Grow: 9,330 Apps Added Last Month, 2010 http://gizmodo.com/5512385/see-android-market-grow-9330-apps-added-last-month
[**IEEEGloss90**]	The Institute of Electrical and Electronics Engineers: IEEE Standard Glossary of Software Engineering Terminology, 1990

	http://www.idi.ntnu.no/grupper/su/publ/ese/ieee-se-glossary-610.12-1990.pdf
[**McGraw04**]	Gary McGraw: Software Security, 2004 http://www.cigital.com/papers/download/bsi1-swsec.pdf
[**MS00**]	Microsoft Corporation: Usability in Software Design, 2000 http://msdn.microsoft.com/en-us/library/ms997577.aspx#uidesign_topic2
[**PezzeYoung08**]	Pezze, Mauro & Young, Michal: Software testen und analysieren, Oldenbourg, 2008
[**Spinellis06**]	Spinellis, Diomidis: Code Qualitiy: The Open Source Perspective, Addison Wesley Professional, 2006
[**UniCarMel99**]	Jiantao Pan: Software Reliability, 1999 http://www.ece.cmu.edu/~koopman/des_s99/sw_reliability/