Thomas Kaufmann, BSc.

Analyzing Continuous Integration in an agile Android Project

Master's Thesis

Graz University of Technology

Institute for Softwaretechnology

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

Graz, 2014

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

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.

Graz, ______________________	______________________
Date	Signature

Eidesstattliche Erklärung[1]

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

Graz, ______________________	______________________
Datum	Unterschrift

[1]Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

Many software projects have come into a situation, which is described as *integration hell* by several well-known authors: Multiple developers are working on various components of the software and shortly before a deadline for the release these modules are merged together to the final software product. This phase is called software integration and can cause serious problems and delays – on the one hand there can be conflicts, if the same part of the code was modified by multiple developers, on the other hand the functionality can be affected as well, since the tests, which are responsible to guarantee a certain quality, might have been deactivated short-term.

To avoid these problems more and more teams use a concept called Continuous Integration. In this process of software development, integrations happen multiple times a day due to breakdown of needed functionality in smaller pieces. Furthermore every single integration is tested on a continuous integration server. During this so called *build*, the final software is built, all existing automated tests are executed and the source code is checked for potential bugs and violations against the team's coding standard. In the case of build errors, the concerned developers are notified through a suitable feedback mechanism like e-mail or SMS, so they can start working on fixing the problems momentarily.

Catroid is an Android application, which provides an opportunity for children and teenager to learn the concept of programming in a playful manner with the language Catrobat and to create small games and multimedia applications without any previous knowledge. This project is developed by voluntary software engineers at the University of Technology in Graz since 2010. Continuous Integration was introduced to the engineering process in 2011 and soon become a key practice. However, by and by first problems arose: the build times increased little by little and soon it took over two hours for one build to succeed. Thus a major advantage of continuous integration was not present any more – the instant feedback. Other problems were related to the slow testing on the Android test device, which was connected to the continuous integration server.

The goal of this thesis is to find out, if appropriate actions were taken to bring the whole continuous integration process back to a stable state as well as discussing possible solutions to increase the support of the development team. To run multiple builds in parallel, three servers were installed, which are working according to the master–slave concept. In order to decrease the time it takes to run a single build, the tests are executed on multiple Android emulators in parallel. Additionally the author of this thesis was responsible to integrate several tools for a static code analysis to the build process to ensure a certain internal software quality.

Zusammenfassung

Viele Softwareprojekte kennen eine Situation, die von mehreren namhaften Autoren als *Integrationshölle* beschrieben wird: Mehrere Entwickler arbeiten an verschiedenen Komponenten der Software, und kurz vor dem Ende der Frist zum Veröffentlichen der Software werden diese Teile zu einem Endprodukt kombiniert. Diese Phase wird Software Integration genannt, in welcher es möglicherweise zu erheblichen Problemen und Verspätungen kommen kann. Einerseits können Konflikte auftreten, wenn gleiche Teile der Software von mehreren Entwicklern verändert wurden, andererseits kann auch die Funktionalität beeinträchtigt sein, weil kurzfristig die Tests, die die Qualität sicherstellen sollten, deaktiviert werden.

Um diese Probleme zu umgehen, verwenden immer mehr Teams ein Konzept namens Continuous Integration. In dieser Methodik der Softwareentwicklung werden durch eine Unterteilung der benötigten Funktionalität in kleinere Stücke, mehrmals am Tag Integrationen durchgeführt. Darüber hinaus wird jede einzelne Integration auf dem sogenannten Continuous Integration Server getestet. Im Laufe dieses sogenannten *Builds* wird das Endprodukt gebaut, alle vorhandenen Tests werden ausgeführt und der Quellcode wird auf eventuelle Fehler und Verstöße gegen die teaminternen Richtlinien überprüft. Sollte es bei einem Build Probleme geben, werden die betroffenen Entwickler durch geeignete Feedback Mechanismen wie E-Mail oder SMS informiert, um sofort mit der Behebung der Fehler beginnen zu können.

Catroid ist eine Android Applikation, die es Kindern und Jugendlichen ermöglicht, mit Hilfe der Sprache Catrobat spielend Programmieren zu lernen und ohne Vorwissen Spiele und Multimedia Applikationen zu erstellen. Dieses Projekt wird an der Technischen Universität Graz von einer Vielzahl an freiwilligen Softwareentwicklern seit 2010 programmiert und betreut.

Continuous Integration wurde 2011 in den Entwicklungsprozess integriert und wurde schnell eine wichtige Komponente für das Entwicklungsteam. Doch mit der Zeit ergaben sich erste Probleme: die Build Zeiten verlängerten sich sukzessive und überstiegen zeitweise die zwei Stunden Marke. Dadurch war ein großer Vorteil von Continuous Integration, nämlich schnelles Feedback zu erhalten, nicht mehr gegeben. Andere Probleme betrafen das zum Teil relativ langsame automatisierte Testen auf dem mit dem Server verbundenen Android Testgerät.

Ziel dieser Arbeit ist es herauszufinden, ob geeignete Maßnahmen getroffen wurden, um den ganzen Continuous Integration Prozess stabiler zu machen, beziehungsweise Möglichkeiten der Verbesserung zu erörtern, die das Team optimal unterstützen können. Um mehrere Builds gleichzeitig ausführen zu können, wurden insgesamt drei Server in Betrieb genommen, die nach dem Master-Slave Prinzip arbeiten. Zum Verringern der Zeit, die ein einzelner Build braucht, werden die Tests mittlerweile parallel auf mehreren Android Emulatoren ausgeführt. Darüber hinaus war der Autor dieser Arbeit dafür verantwortlich, mehrere Tools zur statischen Code Analyse in den Build Prozess einzubinden, um so die interne Softwarequalität sicherzustellen.

Contents

List of Abbrevations

AAPT	Android Asset Packaging Tool
ADB	Android Debug Bridge
AIDL	Android Interface Definition Language
app	application
CI	continuous integration
CPD	Copy/Paste Detector
CVS	concurrent versions system
FOSS	free and open source software
IDE	integrated development environment
IRC	internet relay chat
OHA	Open Handset Alliance
RCS	revision control system
RSS	really simple syndication
SCM	source code manager
SDK	software development kit
SMS	short message service
SVN	subversion
TDD	test driven development
UI	user interface
VCS	version control repository
VM	virtual machine
XP	extreme programming

List of Figures

Listings

1 Introduction

Nowadays software development is mostly done in teams and is a very complex process. A software project is usually divided in different components, where multiple developers add, delete and modify code independently of each other while working on various modules. At the end of an iteration or just before the end of the project, these software components are merged and the software is built - this process is called software integration. The integration process can be very time consuming and according to Duvall [13, p. xx], this kind of late integration can not only increase costs, but also causes project delays. Another term which is used to describe this problem is *integration hell* [14].

Agile software development resulted from the frequent change in requirements during the software engineering process between 1990 and 2000. In contrast to traditional software development, iterations during the development process in agile teams are much shorter. One of the most frequently applied agile development method is extreme programming (XP), which includes a key practice named continuous integration (CI) – this is a process, where developers make small changes, and each of these changes triggers an immediate integration. With this approach an integration should not lead to project delays, because it is done several times a day and integration becomes a non-event [15].

1.1 Motivation

Catroid[1] is an Android[2] application (app) developed by members of the Catrobat project[3] at Graz University of Technology. When development started in 2010, several agile development practices like test driven development (TDD), clean code and pair programming were applied right from the beginning. CI was introduced in November of 2011 and soon became a key practice. A CI server was installed, a test device was connected and every time a developer committed to the main development branch of the version control repository (VCS), the whole application was built, deployed on the device and all automated tests were executed. Every developer could easily check the project's health status by browsing the CI server's web interface. However, as the team as well as the code base have grown, the process has become less flexible than in the beginning by increased build time and wasted time due to waiting queues on the integration server.

Enabling more CI techniques for higher code quality, introducing better feedback mechanisms and showing possibilities to reduce the run time of an integration build are the main goals of this thesis. The following questions should be answered:

- What problems emerged during development using continuous integration?
- What key practices of CI have helped to make the whole development process more stable?
- What can be done to optimize CI for Catroid?

[1]https://github.com/Catrobat/Catroid; last visit: 2014-02-28
[2]http://www.android.com/; last visit: 2014-02-14
[3]http://developer.catrobat.org/; last visit: 2014-02-28

1.2 Structure of this thesis

The chapter on related work (chapter 2) presents an overview of the literature dealing with *Continuous Integration*. Several definitions of CI are discussed as well as the review of different software projects, which apply CI. Additionally a short introduction into both the Catroid and Android project can be found in this chapter.

The next chapter (chapter 3) provides a closer look into the theory of CI. It is shown, which components a typical continuous integration system usually consists of. Furthermore the benefits of CI are discussed and some of the best practices for an efficient continuous integration process are presented. In the latter part of this chapter, a short introduction into software testing in agile teams is given, because testing is a fundamental part of CI.

chapter 4 describes the evolution of the CI process in the Catroid project. The whole process has undergone several changes in the last two years, because the project itself got more and more complex.

The following chapter (chapter 5) explains Martin Fowler's ten best practices on CI and how they are applied in Catroid's continuous integration process.

The last two chapters include an outlook for future tasks in Catroid's continuous integration process and a conclusion. A key concept of CI is rapid feedback on the project's state – due to the growing number of tests, feedback time also increases by and by. Especially in Android teams it is difficult to have a good mix of fast build times and good test coverage. Some counter strategies for these problems are presented, but they require either a general refactoring iteration or a switch to a slightly different integration process. However, both of these CI adoptions are pretty difficult to achieve without hindering the current development progress.

2 Related Work

In this chapter the most important literature dealing with continuous integration is presented, starting with a historical overview of CI, giving several definitions of CI and showing research papers with results of CI in practice. Furthermore a quick introduction in Catroid and Android is given for a better understanding of the next chapters.

2.1 Continuous Integration Definition

While the practice of frequent integrations had been applied for several years before, the term *continuous integration* was formed in the book *Extreme Programming Explained* [2] in 1999. Beck explains CI as a practice, where a programming pair sits down on a dedicated machine, checks out the latest version from the team's source code repository, integrates their own code and runs all the tests. If the developers can not get 100% of the tests passing, the changes have to be reverted, they leave the integration machine and start the development cycle on their own machine again. In case all the tests passed, they commit the code to the VCS [2]. Over the years most software development teams using CI modified this workflow and used a CI server instead of sitting down on a dedicated machine. In the new version of the book Beck explains this asynchronous style of CI, where a continuous integration system detects any changes pushed to the mainline, starts an integration build and automatically runs all the tests [3].

Fowler [14] wrote and rewrote [15] an article about CI which is cited by nearly all articles and journal papers about continuous integration. The most known explanation of CI was published in the updated version [15]:

> *"Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible..."*

One important aspect is the number of integrations – developers should be working on small tasks, which do not take more than a couple of hours to complete. Only with small tasks it is possible to integrate multiple times a day. In the updated article Fowler lists ten key practices for CI. It is not required to adopt each of these techniques in order to perform CI, but applying them will have an enormous impact for the development team working with CI.

The most important book about CI is *Continuous Integration* [13]. Based on the ideas of Fowler several CI practices regarding the CI process itself, continuous testing, feedback mechanisms and code inspections are presented.

2.2 Continuous Integration applied

There are several papers and articles available that deal with the topic *continuous integration*. Some of them show, how CI was introduced in the development process, while others present results of CI in action.

Bowyer and Hughes reflect the use of a CI server in a course for students at the University of Dundee [5]. Results showed a good participation of the students, who extended their knowledge about CI, TDD and configuration management. Hembrink and Stenberg [17] present results of a similar project – continuous integration was used in a course at the Lund University, where students developed a given project using extreme programming. With a limited time span, they had to support the teams with configuration management, so the students could focus on programming, testing and using the tools instead of installing and configuration. Most of the teams had an improved test coverage and low number of failed builds during this course.

Stolberg [29] describes the introduction of a CI system in a running software development team. Getting CI to work took a while, but the retrospective after a few iterations showed major success in the ability to develop and test simultaneously. Miller [23] shows how a CI server can decrease the time spent integrating compared to a manual approach. In this paper several very interesting metrics like broken build reason, length of build breaks and time it takes to fix a build are discussed. He describes a build break as one or more following CI build errors in a row, and shows several possible reasons for broken builds like compilation errors, broken unit tests, failed static analysis and server related issues. When analyzing a CI system, looking at the different build failures and broken build periods might be a good starting point. Miller also mentions that developers might commit more than one task in a single changeset. According to him, this is not a good idea, because a broken build could be more difficult to fix due to the larger changeset. Furthermore a lower frequency of commits increases the probability of merge conflicts.

At the end of the paper, Miller describes his view of the three best practices for development teams using continuous integration [23].

- A build script should not only mark the build as broken in case of an error, but also on warnings.
- The person, who broke the build, has to restore it.
- Developers should not leave after the last commit of the day – they should wait for the result of the CI server, in order to be able to fix a possible build error immediately.

Lacoste [18] demonstrates the results of a CI server experiment, where the integrations rapidly increased. It is explained also that offering developers the possibility to run the whole tests on individual branches on the CI server was a key success factor of the continuous integration process. A complete overview of an CI process from choosing a server to build feedback is given by McGregor [20]. Visible feedback was the key to success to reduce broken builds, because competitive thinking amongst developers to not break the build was promoted. Additionally the code quality went up, because developers added more and more automated test sets.

Abdul and Fhang [1] discuss the impact of CI to a software development team. Due to the fact, that every single commit triggers a build, a new version of the product is available almost immediately after a feature has been integrated. This circumstance is very important for the whole management, because the managers can easily see the changes. Developers benefit from CI as they do not have to spend as much time integrating different modules as within late integration [1]. While most papers about continuous integration focus on the CI server as well as on the executed tests, Souza Pereira Moreira et al. [28] highlight the extraction of metrics after source code analyzation during a CI build. Due to the frequent builds within a CI environment it is relatively easy to visualize a long term analysis for metrics like *number lines of code*, *number lines of comment* and *percentage code coverage of unit tests*.

Vodde [30] presents an introduction to a tool, which measures the capability of continuous integration. This so called *CI grid* visualizes several metrics like compilation, unit testing, test coverage and feedback time for given projects using CI. With help of this table it is easy for the person responsible for CI to detect any bottlenecks within the CI process.

Rogers [25] focuses on the CI environment problems, that occur when a project is getting larger in both size of the code base and size of the team. Since CI requires a well sized test suite, the build takes more and more time. Several counteractions are presented: after agreeing on a maximum build length, the whole build process should be split into different phases. Not all the unit tests, acceptance tests and code analyzers should be run in every single build. All developers require fast feedback for all their changes on the code base. Due to that, Rogers recommends to split the build and separate fast unit tests from longer running acceptance tests. While for the developers unit tests have the highest priority, for the project's management less frequent feedback from the acceptance tests is essential. Otherwise the build time would reach a length, where developers would not be able to commit as often as needed for a smooth CI iteration. Finally the developers should try to write faster unit tests instead of long running acceptance tests when possible. Additionally Rogers [25] illustrates that the frequency of code generation increases by adding new developers to an existing team. However, the larger the team, the more developers are dependent on a build which runs without any errors. If there are any build errors – for whatever reason – all developers who are not involved in fixing the build, are not allowed to commit any changes to the VCS and have to wait, until the person responsible for the broken build has successfully fixed the build.

2.3 Catroid

Catroid is a free and open source software (FOSS), allowing children from the age of eight years to create games and multimedia animations with the visual programming language Catrobat on both Android smartphones and tablets. The lego like programming style lets children and teenager explore paradigms of programming in a playful manner. Catroid is inspired by Scratch[1], a visual programming environment for computers, but as explained in [26], there are some major differences. Instead of using a mouse like on a desktop program like Scratch, on a smartphone the users have to use their fingers to navigate and interact with an application like Catroid. This fact is important for developers and designers of the Catroid project, because it is crucial to provide a simple and easy to use user interface (UI) even on a limited screensize. Since on mobile devices there are several sensors available, some of these sensors can be used in Catroid programs as well. Sensors available on Android phones include [11]:

- Motion sensors like accelerometers and gravity sensors
- Environmental sensors like air temperature and pressure
- Position sensors like magnetometers and orientation sensors

Additionally users can control Parrot's AR.Drones[2], Lego Mindstorm robots[3] and Albert robots[4] via Catroid.

Some other projects within the Catrobat umbrella project are tight coupled with Catroid. With Pocket Paint[5], a Catroid user is able to draw and modify images and users have the possibility to upload, download and remix Catroid projects on a community website[6]. While Catroid[7] has already been published under the name *Pocket Code* in the Android Playstore,

[1]http://scratch.mit.edu/; last visit: 2014-02-28

[2]http://ardrone2.parrot.com/; last visit: 2014-02-28

[3]http://www.lego.com/en-us/mindstorms/; last visit: 2014-02-28

[4]http://www.tsmartlearning.com/en/albert?; last visit: 2014-02-28

[5]https://play.google.com/store/apps/details?id=org.catrobat.paintroid; last visit: 2014-02-28

[6]https://pocketcode.org/; last visit: 2014-02-28

[7]https://play.google.com/store/apps/details?id=org.catrobat.catroid; last visit: 2014-02-28

other team members develop versions for IOS, WindowsPhone, an HTML5 version and a Scratch2Catrobat converter. [27]

2.4 Android

Android is both an operating system for mobile devices and a software platform, and is developed by the Open Handset Alliance (OHA) under the responsibility of Google. A huge variety of different devices is supported, from small smartphones to large tablets. These devices do not only differ in screensize, but also in the presence of different hardware sensors. Since the first version of Android (1.0 *Base* in late 2008) the operating system and user interface have changed drastically. For Android developers it is essential to support as many different clients as possible to achieve a high reach. This goal can be very tricky to reach, because some version steps introduced a numberless amount of new features.

Internally the Android operating system is based on a Linux kernel, which is responsible for hardware interaction, memory management and process management. Although most Android applications are written using the Java programming language, they are not executed in a traditional Java virtual machine (VM), but use the Dalvik VM. This virtual machine is optimized for mobile devices, and allows to run multiple instances. Every single Android application runs in its own process within its own Dalvik instance [21, pp. 14-15]. This feature sandboxes every application and application data cannot be altered from other processes and applications.

Figure 2.1 shows an overview of the Android software stack. On top of the Linux kernel, Android relies on several C/C++ libraries, which provide database support, support for audio and video data, graphic libraries and support for web browsers and internet security. The application framework

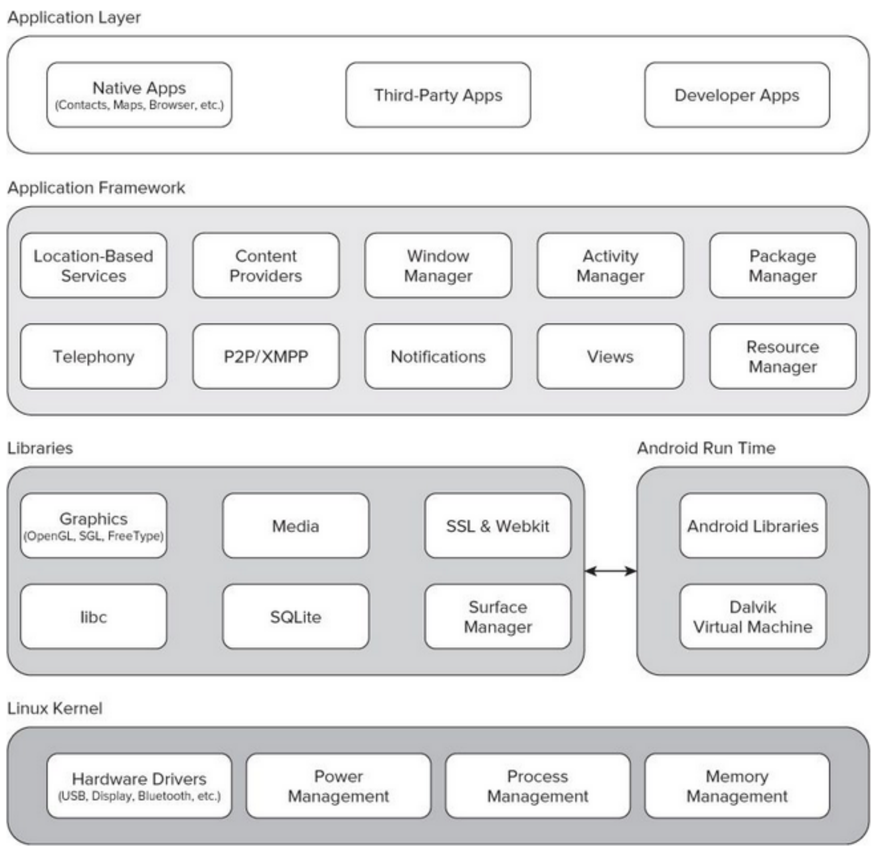


Figure 2.1: Android Software Stack [21, p. 16]

on top of the libraries provides all classes and interfaces needed for Android development and manages the UI as well as application resources. The stack's top tier is the application layer, which includes all applications running on an Android device [21, p. 15].

The Android Runtime includes the Dalvik virtual machine. In order to run programs within this VM, common Java .class files have to be transformed to .dex files with a special tool from the Android software development kit (SDK) called *dx*. Furthermore the Android Runtime includes a set of

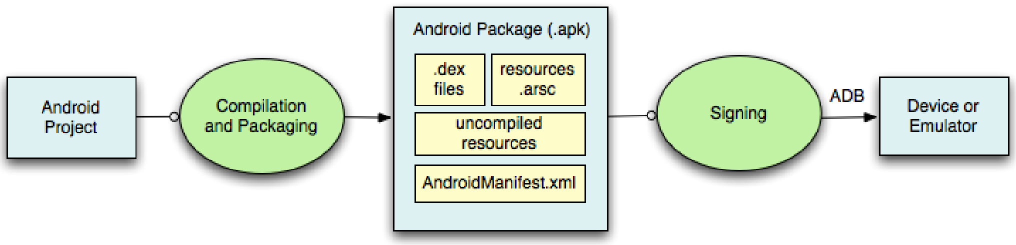


Figure 2.2: Android Build simplified [9]

core Java libraries, which is a subset of the available classes of Java SE. [32, pp. 8-9].

From the development point of view an Android app consists of a variation of Activities, Services, Broadcast Receivers and Content Providers. An Activity is a single screen, which presents the UI and on which the interaction with the user is taking place. With Content Providers it is possible to allow other applications to retrieve, insert, modify and delete internal data of another application. Services enable users to execute long running tasks without blocking the user interface. Broadcast Receivers receive system events like *boot complete* or *battery low*, and enable developers to react to these very specific situations accordingly within an application [8].

The build process has a very high priority within a continuous integration system. The goal is to have presentable software all the time. The result of Android development is a single file called Android package or .apk file. This file can be run on an Android emulator or Android device after it has been signed. The build process of an Android app is very different to the one of other software. Figure 2.2 shows a simplified Android build process. All the tools, which are involved in building an .apk file, are shown in Figure 2.3. Android Asset Packaging Tool (AAPT) compiles all the application resources like the manifest file and layout .xml files, and also generates a R.java file to be able to reference the resources from within the Java code. Android Interface Definition Language (AIDL) compiles all necessary .aidl interface files. The next step is executed with the Java compiler. It takes all the .java files, the compiled .aidl interfaces and the R.java file and compiles them to .class files. The *dex tool* converts all .class files as well as all third party libraries to .dex files, which are needed by the Dalvik VM. The *apkbuilder* uses all compiled and non-compiled resources and all .dex files to generate an .apk file. This file is not ready to be loaded onto the emulator or a physical device yet, because it has to be signed with the *Jarsigner* – either with a debug key for development or with a release key for production in Google's PlayStore [9].

Fortunately, developers do not have to execute all the steps from above by hand. On their development machines, they are assisted by integrated development environment (IDE)s, which simplify the process to only a few mouse clicks. Furthermore it is possible to use common build tools to build Android applications. For several years, Ant[8] was the standard and it was possible to build and sign an .apk file with a single command. However, in 2013, Android introduced Gradle for Android[9] to offer all Android developers more flexibility when building their applications.

[8]http://ant.apache.org/; last visit: 2014-04-16

[9]http://www.gradleware.com/android/gradle-the-new-android-build-system/; last visit: 2014-04-16

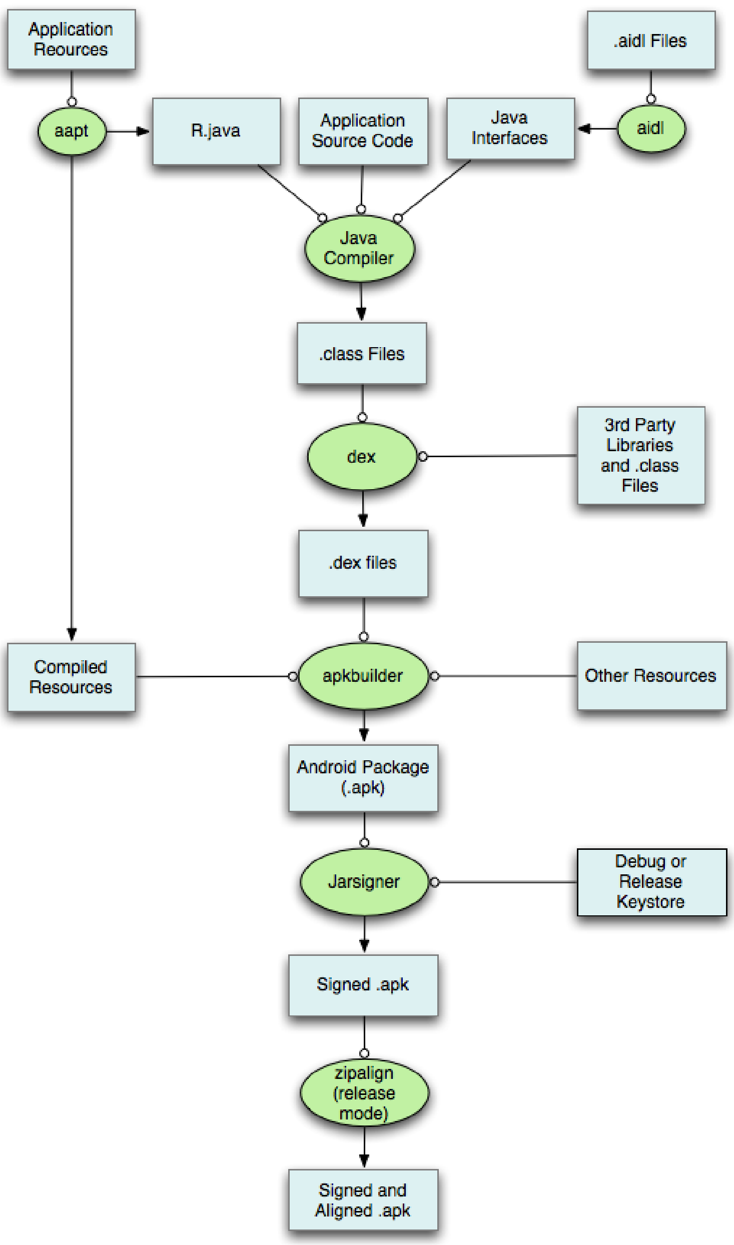


Figure 2.3: Android Build Process [9]

3 Theoretical Background

3.1 Components of a Continuous Integration System

A development team is not applying continuous integration if the developers are only using a test server to run automated software tests. CI is more than this – it is a process with frequent interaction between people, hardware and software. Figure 3.1 illustrates how these parts of a CI system cooperate [13, p. 5].

1. developers add, delete or modify files and commit the changes to a VCS
2. a CI server running on a separate integration build machine is polling for changes in this repository
3. the last commit is detected, the CI server checks out the latest version of the source code and starts a build script for the integration
4. the build script is responsible for compiling the source code, executing all automated tests, running static code analysis tools and deploying the software
5. after the build has finished, developers are notified about the state of the integration and the CI server starts polling for changes again

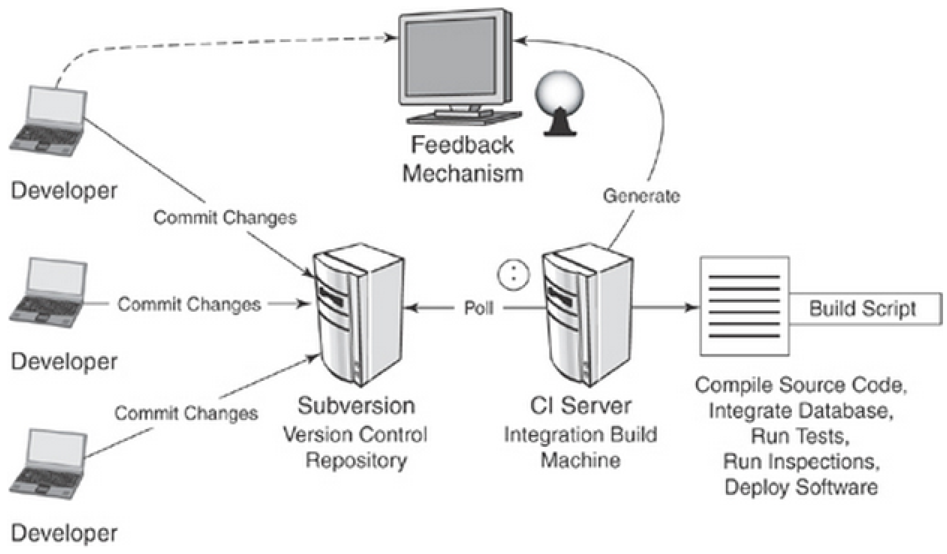

Feedback Mechanism
Developer
Commit Changes
Generate
Commit Changes
Poll
Build Script
Developer
Commit Changes
Subversion Version Control Repository
CI Server Integration Build Machine
Compile Source Code, Integrate Database, Run Tests, Run Inspections, Deploy Software
Developer

Figure 3.1: Components of a CI system [13, p. 5]

Developer

Once a developer has finished all modifications related to the current task a private build is run manually on the developer's local machine. During this build possible changes from other developers in the repository are integrated, and only if the private build is successful the developer commits the changes to the VCS. Although many IDEs assist developers running private builds by providing mechanisms to run tests, check code and commit changes, it should be possible to run a whole private build with a single command from the command-line [13, pp. 5-6].

The term *developer* is not only used for people altering source code, but also includes software testers, quality managers and administrators. Source code is not the only category of files residing in a VCS. Testers can add new tests, quality managers might change rules for source code analyzation and

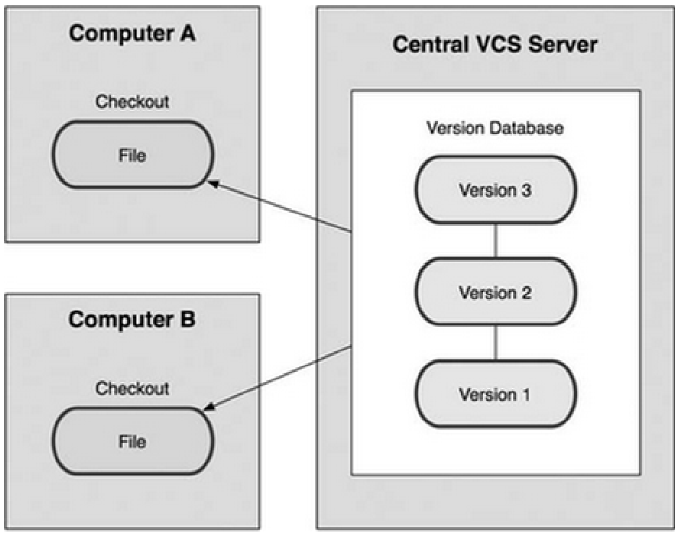


Figure 3.2: Centralized VCS [6]

administrators could change build scripts and update libraries. A CI server should recognize all of these modifications and start an integration build.

Version Control Repository

One of the requirements for using CI is a VCS [13, pp. 7-8]. Other terms used for VCS are source code manager (SCM) and revision control system (RCS). In simple terms, a VCS is a tool that manages changes to source code and records all changes including a date, person and diff [19, p. 1]. The major benefits of using a VCS system are the possibility to revert files and whole projects to previous states, retrieve deleted files and visualize changes [6]. Other advantages of using version control are given by O'Sullivan [24]. Software teams using a VCS are able to track the complete history of the project by retrieving information for every single change – which developer changed which files at what time for which reason. Furthermore a RCS helps involved people to solve possible merge conflicts when changes are made in the same files simultaneously. One major benefit, especially for software development teams, is the possibility to handle multiple versions of a software more easily [24]. A short overview of different version control systems is given by Chacon [6]. A *Local Version Control Repository* helped a single person to restore a previous state of files by keeping the changes in a database. Multiple developers could use revision control by using one of the *Centralized Version Control Systems*. All files in a project are stored in a server, and clients are able to check out the needed files from it, make modifications and push the changes to the server. These version control systems like concurrent versions system (CVS) or subversion (SVN) have one major downside – if the connection to the server is broken for some reason, it is not possible to push changes. *Distributed Version Control Systems* like Git or Mercurial tackle this problem by cloning the full repository on a checkout – every developer has the complete repository on his own machine.

Figure 3.2 shows a sample centralized VCS. The whole history of the only sample file residing in the repository is on the central server, but both clients can checkout the latest version of this file. In contrast to this scenario, Figure 3.3 shows a distributed VCS, where each client has the whole history of the file cloned after a checkout. Due to this fact it is possible to view the history even without an active connection to the server, or restore all files if the server repository has been broken for any reason.

Integration Build Machine

An integration build machine is a separate machine hosting the CI server (see CI Server) [13, p. 12].

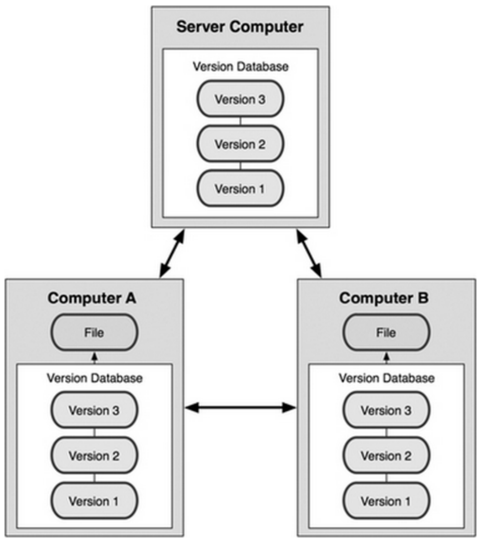


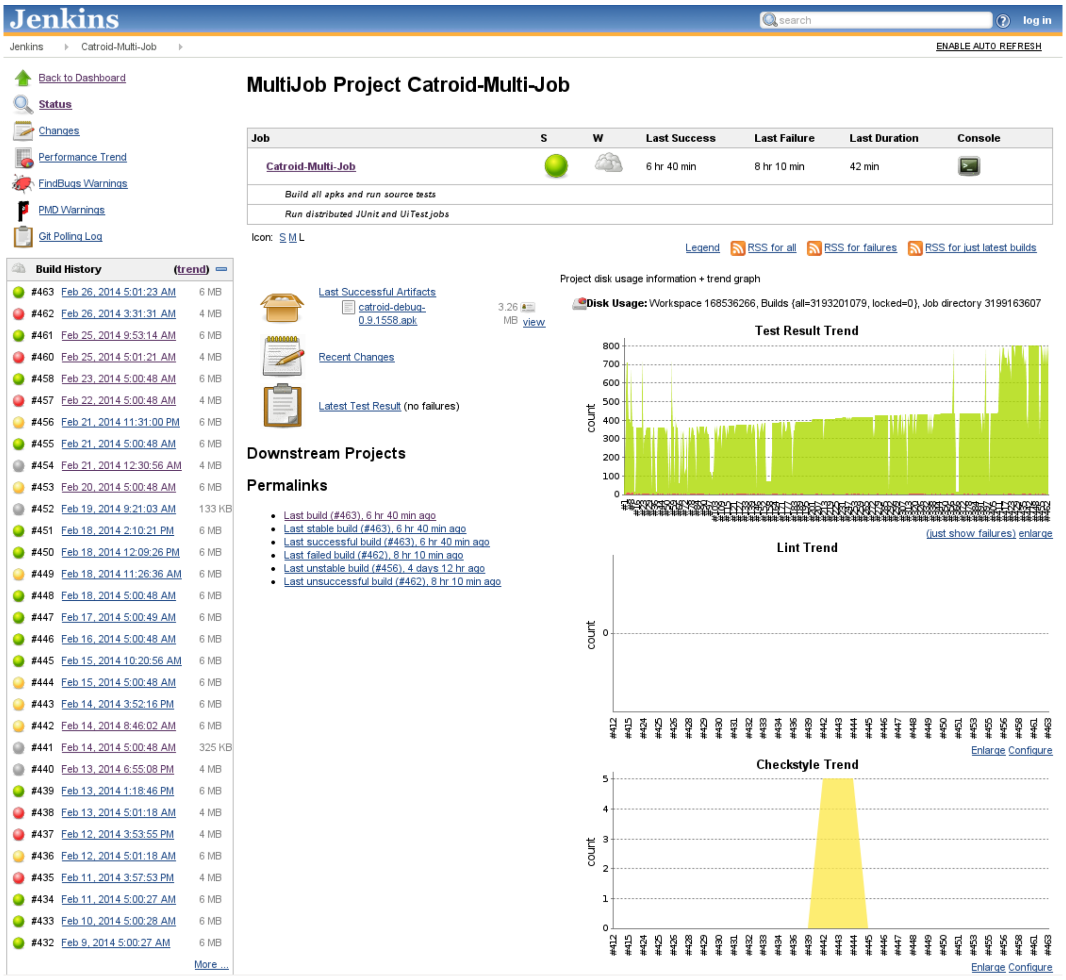
Figure 3.3: Distributed VCS [6]

CI Server

A CI server is a software tool which checks a VCS (see section 3.1) for changes and starts a build script for integration when changes are detected. The server should be configured to check for changes in a VCS at least every ten minutes [13, pp. 8-9]. Usually a CI server uses a web interface to show build results and offer the possibility to download latest artifacts. However, in this thesis the primary focus regarding the CI server is on Jenkins[1] , an open source continuous integration server.

Figure 3.4 shows a Jenkins' web-interface of a particular job. On the left side the latest builds including state symbols and starting times are displayed, while on the right side test and code analysis trends are visualized. Within Jenkins there exist four different kinds of build results:

[1]http://jenkins-ci.org/; last visit: 2014-03-03



MultiJob Project Catroid-Multi-Job

Job	S	W	Last Success	Last Failure	Last Duration	Console
Catroid-Multi-Job	●		6 hr 40 min	8 hr 10 min	42 min	
Build all apks and run source tests						
Run distributed JUnit and UiTest jobs						

Build History (trend)

●	#463	Feb 26, 2014 5:01:23 AM	6 MB
●	#462	Feb 26, 2014 3:31:31 AM	4 MB
●	#461	Feb 25, 2014 9:53:14 AM	6 MB
●	#460	Feb 25, 2014 5:01:21 AM	4 MB
●	#458	Feb 23, 2014 5:00:48 AM	6 MB
●	#457	Feb 22, 2014 5:00:48 AM	4 MB
●	#456	Feb 21, 2014 11:31:00 PM	6 MB
●	#455	Feb 21, 2014 5:00:48 AM	6 MB
●	#454	Feb 21, 2014 12:30:56 AM	4 MB
●	#453	Feb 20, 2014 5:00:48 AM	6 MB
●	#452	Feb 19, 2014 9:21:03 AM	133 KB
●	#451	Feb 18, 2014 2:10:21 PM	6 MB
●	#450	Feb 18, 2014 12:09:26 PM	6 MB
●	#449	Feb 18, 2014 11:26:36 AM	6 MB
●	#448	Feb 18, 2014 5:00:48 AM	6 MB
●	#447	Feb 17, 2014 5:00:49 AM	6 MB
●	#446	Feb 16, 2014 5:00:48 AM	6 MB
●	#445	Feb 15, 2014 10:20:56 AM	6 MB
●	#444	Feb 15, 2014 5:00:48 AM	6 MB
●	#443	Feb 14, 2014 3:52:16 PM	6 MB
●	#442	Feb 14, 2014 8:46:02 AM	6 MB
●	#441	Feb 14, 2014 5:00:48 AM	325 KB
●	#440	Feb 13, 2014 6:55:08 PM	4 MB
●	#439	Feb 13, 2014 1:18:46 PM	6 MB
●	#438	Feb 13, 2014 5:01:18 AM	4 MB
●	#437	Feb 12, 2014 3:53:55 PM	4 MB
●	#436	Feb 12, 2014 5:01:18 AM	6 MB
●	#435	Feb 11, 2014 3:57:53 PM	4 MB
●	#434	Feb 11, 2014 5:00:27 AM	6 MB
●	#433	Feb 10, 2014 5:00:28 AM	6 MB
●	#432	Feb 9, 2014 5:00:27 AM	6 MB

More ...

Last Successful Artifacts
catroid-debug-0.9.1558.apk 3.26 MB view

Recent Changes

Latest Test Result (no failures)

Project disk usage information + trend graph

Disk Usage: Workspace 168536266, Builds {all=3193201079, locked=0}, Job directory 3199163607

Downstream Projects

Permalinks

- Last build (#463), 6 hr 40 min ago
- Last stable build (#463), 6 hr 40 min ago
- Last successful build (#463), 6 hr 40 min ago
- Last failed build (#462), 8 hr 10 min ago
- Last unstable build (#456), 4 days 12 hr ago
- Last unsuccessful build (#462), 8 hr 10 min ago

Figure 3.4: Catrobat Jenkins Server - Job overview

- Green build – this state indicates a successful build
- Yellow build – compilation was successful, but at least one test or an inspection failed
- Red build – if a build's state is red, most likely a compilation error occurred or other fatal problems during the test run failed the build
- Grey build – build timed out or was aborted

If the developer team does not use any additional feedback mechanism, the developers most likely use the web-interface to check the health status of a certain project. Thus a page like in Figure 3.4 is the first place to go to search for a specific build.

Build Script

A build script can be a single script, or a set of scripts, and is executed by the CI server if a change in the repository is detected. This build script usually checks out the latest changes from the VCS, compiles the code, executes all the tests, runs code inspections and deploys the software. Build scripts are very common in software development and are not only used by CI servers. Usually developers are assisted by IDEs during development on their local machines by providing their own build scripts for compilation and running tests. In a CI environment, the build script should be completely decoupled from any IDE, so it is possible to run the build script without having a certain IDE installed [13, p. 10].

Feedback Mechanism

Without feedback CI can not work. If an integration build has failed team members should be informed immediately to fix the problem as soon as possible [13, p. 10]. There are many possible ways of generating feedback like e-mail notification, sending short message service (SMS), writing really simple syndication (RSS) feeds or posting results to channels in internet relay chat (IRC). What channel should be used and which team members are informed should be adjusted from team to team and is also dependent on the capability of the CI server.

3.2 Benefits of Continuous Integration

The introduction of a CI system into a stable running development process can be very time consuming. It has to be of verifiable value to switch to CI. In his book *Continuous Integration mit Hudson* Wiest presents several benefits of CI [31, pp. 23ff].

Reduced Risks

Due to frequent and fully automated integrations a software team can avoid unpleasant surprises at the end of an iteration [31, p. 23]. Fowler describes integration as a long, unpredictable process – it can take up to several months to complete, and during the integration process nobody can tell when it is going to be finished. By turning integration to a *non-event* due to frequent integration it is easy to solve integration errors, since the code base has diverged at most for some hours [15]. In case of a failing integration build the whole developer team is responsible for correcting the error as soon as possible.

Duvall describes reduced risks in detail[13, pp. 29f]. In a CI system, integrations, tests and inspections are run multiple times a day – thus there is a high probability that defects are recognized at an early stage instead at the end of an iteration. With a CI server, integration is done in a clean environment in a same way again and again. This leads to reduced assumptions like needed libraries or environment variables for a build.

Better Product Quality

By applying CI there is a high probability that the quality of the software is higher than the product quality of traditionally developed software. The main goal is that the end product is containing of as little errors as possible. With the use of a CI server this goal is achieved by running all the tests on every single change in the VCS. Errors are also detected much earlier, because test results are presented after every small change, while in traditional software development tests are often executed at the end of an iteration. Furthermore the code coverage is generally higher on a CI server with automated tests than with a manual approach [31, pp. 24f].

Having presentable Software all the Time

A software team running CI builds always has a running software product. With every new commit to the mainline the latest source code is compiled, tests and code inspections are run. The compiled software might not be perfect as far as functionality is concerned, but for a successful integration build all tests and code analyzation have to pass. This generated product can be used by several team members for demonstration purposes [31, pp. 24f]:

1. testers have access to up-to-date versions
2. clients can see the development progress easily
3. project managers are able to spot problems earlier

Better Efficiency

A precondition for continuous integration is an automated build process. An automated build is always faster than a manual building of a software product step by step. With an automated build script it does not matter whether the software is built on the CI server or on a developer machine. This possibility makes it easier to instruct new team members by showing them how to run a private build on their own machine with a single command [31, pp. 26f].

Documented Build Process

In a CI system, all build steps are recorded in the server configuration and in build scripts. This kind of documentation is executed on every single integration and can be observed on the web-interface during a build on the CI server [31, p. 27].

Higher Motivation

More integrations produce more feedback for the developers. Most of the time, this feedback should be positive, indicating that the changes pushed to the CI server were integrated successfully and the project's health status did not switch from green to red. On the other hand, even negative feedback, indicating a failing testcase for example, is positive, if the developer responsible for the failure, is immediately working on fixing the build. In order to get feedback more frequently, developers tend to split their tasks into many small tasks over the time, and commit each of these sub tasks. Therefore it is possible that they get frequent feedback from the CI server multiple times a day, and their confidence gets boosted, because they can see the project progress [31, p. 28].

Better Information Visibility

Due to the fact, that software is built and tested fully automatically on every single change, within a CI system, information regarding the software product is always up to date. Many CI server present several metrics like code coverage, test results, code inspections and build time in a graphical manner. Thus it is easy for everyone to check these metrics [31, p. 29]

Supported Process Enhancement

By using CI it is possible to gather important metrics for the development process [31, p. 29]. Increased build times and hardware bottlenecks like low storage and missing servers can be easily identified and fixed.

3.3 Practices in a Continuous Integration Process

Duvall describes seven practices of CI which should be applied to benefit from continuous integration [13, pp. 39-43]. These practices are not connected with CI and can be applied by teams without a CI server as well, but within a continuous integration process they unlock their potential.

Commit Code Frequently

Only by committing changes frequently it is possible to get feedback from the CI server as quick as possible with a limited chance of breaking the build. If the changeset is too big and either the feature branch or the mainline branch have undergone several changes, there is a high probability that an integration build on the CI server will fail. Duvall shows two possible strategies for frequent commits:

- Small changes: developers should work on very small tasks and should only write the tests and necessary source code to fix these tests, and commit their changes afterwards.
- Commit after each task: it should be possible to finish a task within a few hours. If this goal cannot be achieved, the bigger task has to be split into several smaller ones. Developers should aim for a commit after each of these *sub tasks* and should not wait until all sub tasks are finished.

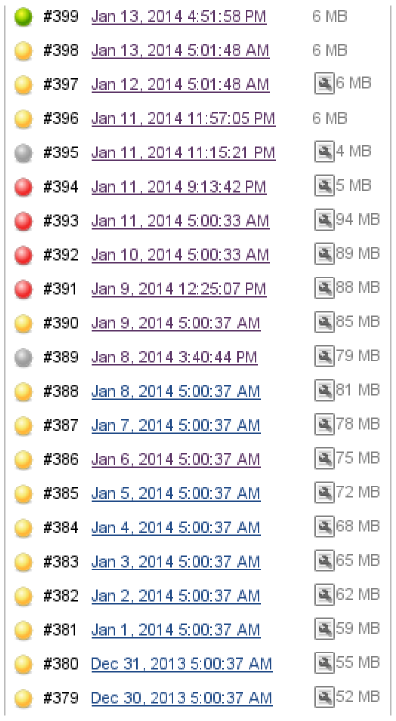
Furthermore it should be avoided, that all developers commit their changes at the same time. This would lead to many broken builds. If that scenario happens at the end of the day, it could be troublesome on the next day, because not only one build error might have to be fixed, but also several possible errors are existing in the latest build [13, pp. 39f].

Don't Commit Broken Code

Changes that do not work should not be committed to the VCS. A private build can prevent broken code getting committed [13, p. 41]. However there are situations where the integration build on the CI server is failing even after a successful private build – for example if a developer uses a new library and does not add it unintentionally to the VCS. This missing library should lead to a compilation error on the CI server.

Fix Broken Builds Immediately

From time to time a build fails on the integration machine. Duvall gives several examples for broken builds like compilation errors, failed tests, failed



🟢	#399	Jan 13, 2014 4:51:58 PM	6 MB
🟡	#398	Jan 13, 2014 5:01:48 AM	6 MB
🟡	#397	Jan 12, 2014 5:01:48 AM	6 MB
🟡	#396	Jan 11, 2014 11:57:05 PM	6 MB
⚪	#395	Jan 11, 2014 11:15:21 PM	4 MB
🔴	#394	Jan 11, 2014 9:13:42 PM	5 MB
🔴	#393	Jan 11, 2014 5:00:33 AM	94 MB
🔴	#392	Jan 10, 2014 5:00:33 AM	89 MB
🔴	#391	Jan 9, 2014 12:25:07 PM	88 MB
🟡	#390	Jan 9, 2014 5:00:37 AM	85 MB
⚪	#389	Jan 8, 2014 3:40:44 PM	79 MB
🟡	#388	Jan 8, 2014 5:00:37 AM	81 MB
🟡	#387	Jan 7, 2014 5:00:37 AM	78 MB
🟡	#386	Jan 6, 2014 5:00:37 AM	75 MB
🟡	#385	Jan 5, 2014 5:00:37 AM	72 MB
🟡	#384	Jan 4, 2014 5:00:37 AM	68 MB
🟡	#383	Jan 3, 2014 5:00:37 AM	65 MB
🟡	#382	Jan 2, 2014 5:00:37 AM	62 MB
🟡	#381	Jan 1, 2014 5:00:37 AM	59 MB
🟡	#380	Dec 31, 2013 5:00:37 AM	55 MB
🟡	#379	Dec 30, 2013 5:00:37 AM	52 MB

Figure 3.5: Catrobat Jenkins Server - broken build period

inspections or failed deployment. Due to frequent integrations and a short period between the broken build and the last successful one the error can be fixed easily. Fixing a broken build has top project priority [13, p. 41]. A situation like in Figure 3.5 should not happen – the mainline build on the CI server is broken for over two weeks. In these two weeks nobody was able to get a successful private build and nobody should have committed to the mainline except for fixing the build.

Write Automated Developer Tests

Nearly all software development teams automate their unit tests. In order to get the most out of the CI process, they should aim to automate all other tests as well. A test suite might consist of unit tests, component tests, system tests and functional tests. Of course it is not a good idea to run all these tests during every single build. This would lead to a build time that is unacceptable. There are several tools assisting developers in writing automated tests. The most important tool in the *Java world* is jUnit [13, p. 41].

All Tests and Inspections Must Pass

When applying CI, tests and inspections are as important as compilation. All tests must pass for a successful build. Just a single failing testcase shows that there is a problem in code. Automated inspectors can prevent developers from committing code that does not follow coding and design standards [13, p. 42].

Run Private Builds

A private build is an integration build executed on a developer's machine. After the developer has finished his task and ran all unit tests he executes a private build on his local machine. The build script for the private build should get changes from the VCS, compile developer's changes together with the newest code base from the repository and runs both unit tests and inspections. A successful private build will make it less likely that the build on the CI server will fail [13, p. 42]. Figure 3.6 shows the described scenario. A commit to the repository is only allowed after a successful

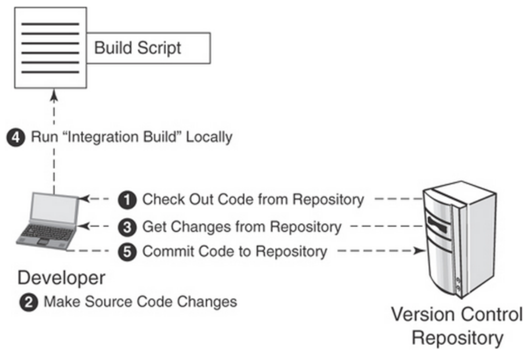

Build Script
4 Run "Integration Build" Locally
1 Check Out Code from Repository
3 Get Changes from Repository
5 Commit Code to Repository
Developer
2 Make Source Code Changes
Version Control Repository

Figure 3.6: Running a private build [13, p. 43]

private build, which includes getting the changes from the repository after local development has been finished.

Avoid Getting Broken Code

If the mainline build on the CI server is broken, developers should not check out the latest version. Since the team is responsible for fixing the build somebody has to be working on the problem already. If developers are working together in a room, a passive feedback mechanism like a traffic light might be more informative than e-mails, because maybe not all developers have read the e-mail sent after a broken build [13, p. 43]. If the build failed due to a compilation error, any developer who checks out the latest version is not able to compile his changes locally.

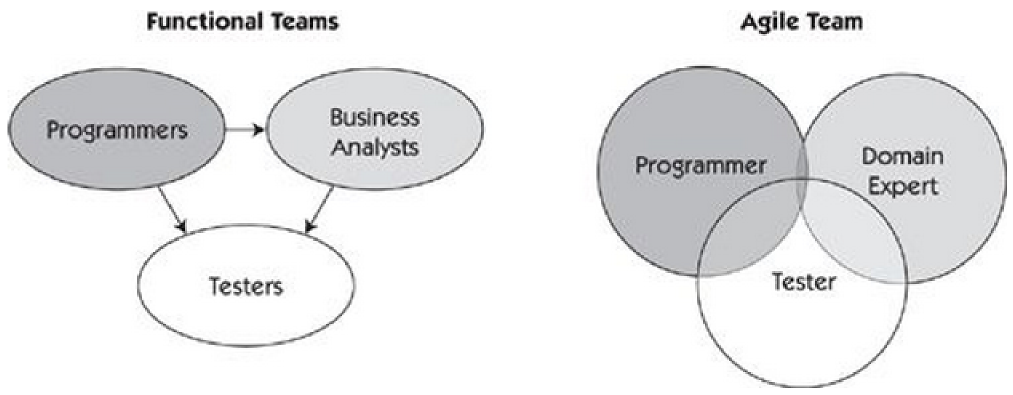


Figure 3.7: Functional Team vs. Agile Team [7, p. 64]

3.4 Software Testing in Agile Teams

Software testing is different when applied in agile teams compared to traditional software teams. Within traditional teams, testing is done after the development – this so called *testing phase* can be a long time span and it can be difficult to increase software quality a long time after the actual code was written. In these long running development cycles, team members try to make sure, that all needed requirements are present in the release of the final product. If a requirement is not ready for the release, the release is usually postponed. However, sometimes the test phase is skipped, because the time is needed in the *development phase* and the final product is not tested sufficiently.

Agile teams use short development cycles. During these short iterations, the team works closely together with the management and the customers, and has a good knowledge of the requirements. While traditional testers (Figure 3.7) usually have to wait for programmers to complete the code before beginning to write tests, within agile teams, programmers should

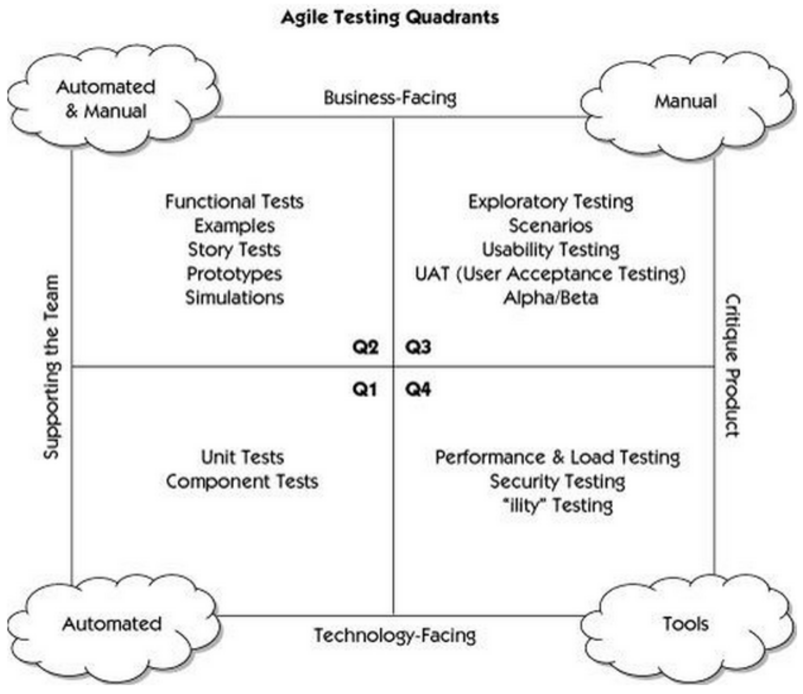


Figure 3.8: Agile Testing Quadrants [7, p. 98]

not get very far ahead of the testers. The reason for this is, that a story is not completed (some teams use the term "*done*") until there exist tests for it. Furthermore agile teams may consist of cross-functional team members with different knowledge backgrounds [7, pp. 9ff] [7, p. 64].

Crispin and Gregory introduce a diagram dividing relevant test types in four quadrants – on the one hand tests, which support the team, and on the other hand tests, that test the product in detail. The second axis divides tests into business-facing and technology-facing tests. Figure 3.8 shows this diagram, with the left quadrants (Q1, Q2) including tests supporting the team during development of the product, and the right quadrants (Q3, Q4) critiquing the product [7, pp. 97f].

Quadrant 1

Q1 covers both, unit tests and component tests, which are also main integral parts of test-driven development [7, p. 99]. Beck explains a general TDD cycle, which consists of the following three steps: [4, p. 11].

1. First of all, a test has to be written. Executing this test leads to a failure, because the code has not been implemented yet.
2. Production code is written in order to make the test passing. The code does not need to be perfect – its only business is a working test.
3. The code is refactored in this step. Duplications are removed and the code should be made more readable.

Usually, unit tests cover a small subset of the whole system like a class or a method, and component tests verify the behavior of a set of classes. Both test categories are usually automated with an automation tool from the xUnit family. Due to the fact, that these tests are written in the same language as the production code, the team's business members might have difficulties to understand them. If the developer team uses an automated build process like continuous integration, tests from this quadrant should be executed on every single code change to provide rapid feedback [7, p. 99].

Quadrant 2

Similar to the tests in Q1, the tests from Q2 also assist the work of the whole development team. Crispin and Gregory specify these tests as business-facing tests, customer-facing tests or customer tests. The variety of Q2–tests, which are shown in Figure 3.8, are deducted from customer examples and represent features, which are required by the customer. In contrast to the ones from Q1, the tests from Q2 are written in a programming language, which is easily understandable by business experts. Furthermore

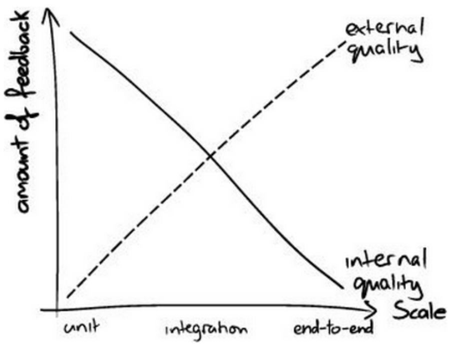


Figure 3.9: Feedback from Different Kind of Tests [16, p. 11]

it is possible, that tests from Q2 are testing similar components as tests in Q1. In order that these so called higher level tests can provide fast feedback as well, a majority of them should be automated and run periodically [7, pp. 99f].

According to Crispin and Gregory, tests from Q1 measure the internal quality, while Q2 tests define external quality [7, p. 99]. Freeman and Pryce use a diagram (Figure 3.9) to show the differences between various kinds of tests. On the one hand, external quality shows, how the system fits from a customer's or user's point of view – for example, if the functionality is correct or if the system is responsive. On the other hand, internal quality is a metric, how well the product meets the needs of the developers – for example, if changes can be applied easily, or if code is easy understandable even for new developers. End-to-end tests (they would reside in Q2) represent the external quality and show, if developers understand the needs of customers and users, but they do not show, if the code is written well. Meanwhile, unit tests present a good feedback about the code itself regarding the quality, and prove that subsystems of the product is working well individually, but these tests can not expose, that the whole system is working correctly [16,

p. 11].

Quadrant 3

Tests in Q3 simulate, how real users would use the software product, and for this reason have to be run manually by human testers. One important part of these tests is usability testing, where alpha, beta and final releases are tested with probands of the target audience of the end product [7, pp. 101f]. Due to the manual approach of these tests, they are not important for a continuous integration process.

Quadrant 4

Crispin and Gregory use the term *technology-facing-tests* for the tests in Q4, which are performed by certain tools in order to test performance, robustness and security – only load&performance tests can be fully automated [7, pp. 102f]. Some CI environments use tests from this quadrant. For example, performance and stress tests could be executed with a nightly build and provide long-term metrics for the developers.

4 Catroid Continuous Integration Process

Continuous integration was introduced to Catroid in late 2011. Jenkins was used as a CI server and was installed on a separate integration machine. A reference mobile device was connected to the server and every commit to the mainline triggered an integration build including compilation and running automated unit tests and functional tests.

4.1 Initial Analyzation

Over a period of 12 months the runtime for a single integration build on the CI server has increased from about 30 minutes to over two hours. Since compilation time has not changed significantly this behavior resulted primarily from adding more and more long running functional tests (see Figure 4.1 for a long term visualization of all tests).

Right from the beginning, some Catroid tests required a bluetooth server – thus, developers had difficulties running private builds on their local machines. Additional problems were the long build time and the UI tests which only worked relatively stable on the test device connected to the CI server. Every task, a developer is working on, is done in a separate branch. To verify that the mainline build would be successful after a merge, the

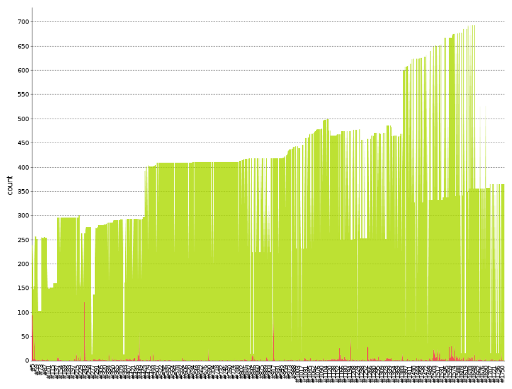


Figure 4.1: Development of Catroid Tests

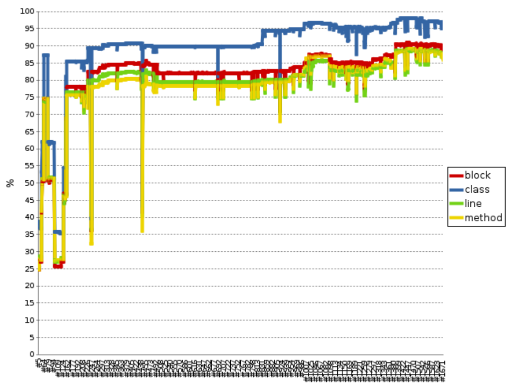
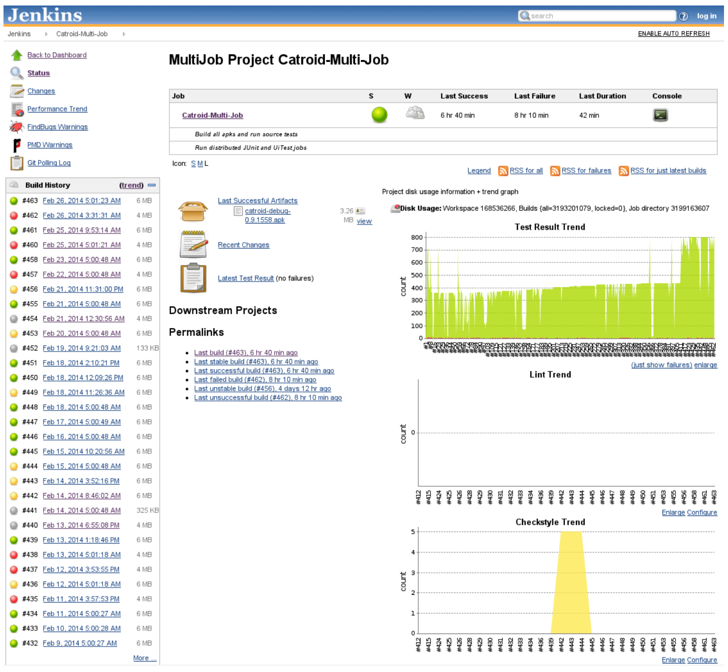


Figure 4.2: Coverage Trend

Catroid team uses a slightly different work flow compared to the traditional CI process. Instead of running a private build on the developer's local machine, Catroid developer work on their tasks locally, commit changes to so called feature branches, push these branches to the VCS and start a job manually on the CI server by entering their branch name – this build is called *custom branch test*. Rather than building and testing the mainline, this job compiles and tests the feature branch on the connected device. This process decreased failed mainline builds significantly, because code was only merged to the mainline in case of a successful build on the feature branch. Unfortunately with just one device connected to the server and due the long build time this practice lead to long building queues and frustrated developers.

Unfortunately, the builds for the master branch failed from time to time for several different reasons. The listing below summarizes these errors and the corresponding build status on the Jenkins server.

- The long runtime of over two hours caused so called *OutOfMemory exceptions* on the test device, and the whole testrun was aborted $\rightarrow$ red build


Jenkins › Catroid-Multi-Job › ENABLE AUTO REFRESH

Back to Dashboard
Status
Changes
Performance Trend
FindBugs Warnings
PMD Warnings
Git Polling Log

MultiJob Project Catroid-Multi-Job

Job	S	W	Last Success	Last Failure	Last Duration	Console
Catroid-Multi-Job	🟢	☁	6 hr 40 min	8 hr 10 min	42 min	
Build all apks and run source tests						
Run distributed JUnit and UiTest jobs						

Icon: S M L

Legend RSS for all RSS for failures RSS for just latest builds

Build History (trend)

🟢	#463	Feb 26, 2014 5:01:23 AM	6 MB
🔴	#462	Feb 26, 2014 3:31:31 AM	4 MB
🟢	#461	Feb 25, 2014 9:53:14 AM	6 MB
🔴	#460	Feb 25, 2014 5:01:21 AM	4 MB
🟢	#458	Feb 23, 2014 5:00:48 AM	6 MB
🔴	#457	Feb 22, 2014 5:00:48 AM	4 MB
🟡	#456	Feb 21, 2014 11:31:00 PM	6 MB
🟢	#455	Feb 21, 2014 5:00:48 AM	6 MB
⚪	#454	Feb 21, 2014 12:30:56 AM	4 MB
🟡	#453	Feb 20, 2014 5:00:48 AM	6 MB
⚪	#452	Feb 19, 2014 9:21:03 AM	133 KB
🟢	#451	Feb 18, 2014 2:10:21 PM	6 MB
🟢	#450	Feb 18, 2014 12:09:26 PM	6 MB
🟡	#449	Feb 18, 2014 11:26:36 AM	6 MB
🟢	#448	Feb 18, 2014 5:00:48 AM	6 MB
🟢	#447	Feb 17, 2014 5:00:49 AM	6 MB
🟢	#446	Feb 16, 2014 5:00:48 AM	6 MB
🟢	#445	Feb 15, 2014 10:20:56 AM	6 MB
🟡	#444	Feb 15, 2014 5:00:48 AM	6 MB
🟡	#443	Feb 14, 2014 3:52:16 PM	6 MB
🟡	#442	Feb 14, 2014 8:46:02 AM	6 MB
⚪	#441	Feb 14, 2014 5:00:48 AM	325 KB
⚪	#440	Feb 13, 2014 6:55:08 PM	4 MB
🟢	#439	Feb 13, 2014 1:18:46 PM	6 MB
🔴	#438	Feb 13, 2014 5:01:18 AM	4 MB
🔴	#437	Feb 12, 2014 3:53:55 PM	4 MB
🟡	#436	Feb 12, 2014 5:01:18 AM	6 MB
🔴	#435	Feb 11, 2014 3:57:53 PM	4 MB
🟢	#434	Feb 11, 2014 5:00:27 AM	6 MB
🟢	#433	Feb 10, 2014 5:00:28 AM	6 MB
🟢	#432	Feb 9, 2014 5:00:27 AM	6 MB

More ...

Last Successful Artifacts
catroid-debug-0.9.1558.apk 3.26 MB view

Recent Changes

Latest Test Result (no failures)

Downstream Projects

Permalinks

- Last build (#463), 6 hr 40 min ago
- Last stable build (#463), 6 hr 40 min ago
- Last successful build (#463), 6 hr 40 min ago
- Last failed build (#462), 8 hr 10 min ago
- Last unstable build (#456), 4 days 12 hr ago
- Last unsuccessful build (#462), 8 hr 10 min ago

Project disk usage information + trend graph

Disk Usage: Workspace 168536266, Builds {all=3193201079, locked=0}, Job directory 3199163607

Figure 4.3: Build Result Catroid Master Branch

- Android Debug Bridge (ADB), the software used by Android to interact between Android devices and a computer, disconnected during testrun → red build
- UI tests were unstable and failed randomly → yellow build
- Custom branch test was not executed and broken code got committed to the mainline → red or yellow build
- Internet connection (Wlan)was lost during testrun → yellow build
- Screen lock on test device was automatically activated during testrun, and the build timed out → gray build

While it is almost impossible to prevent all of these scenarios, the unstable UI tests can be annoying. After a testrun with one or more failing test cases, team members have to have a closer look if these failing tests have failed for the first time, or if they used to fail from time to time in the past. Freeman and Pryce describe these tests as *flickering tests*, and they should be refactored in order to pass every single time as soon as possible – otherwise team members can not rely on the testresults any more and real defects could be overlooked [16, p. 317].

Figure 4.3 shows the different build results over a given time period on the left side. It can be observed that the build status changes after almost every single build – this behavior decreases the advantages of CI, because the involved team members have to have a closer look on nearly every single build instead the possibility to work on the next story in case of a green build.

Within the next sections enhancements are described which made the CI process more efficient and stable.

4.2 Add more Servers

As a first step, two more physical servers were added to the CI infrastructure. Jenkins supports the installation of additional machines by using a master-slave concept. The master server is the one where the CI server instance - including Jenkins' web interface – is running. Additionally all build artifacts like test results and built .apk files are stored on the master server. Most jobs are swapped to the slave machines to keep the web interface on the master node more responsive when running multiple jobs simultaneously. Therefore identical test devices were connected to each of the servers and from then on it was possible to execute three builds, a mix of custom branch tests and mainline builds, in parallel. The waiting queue during main development hours became much shorter and thus the feedback time from the CI server was reduced in case multiple developers started builds simultaneously.

4.3 Make the Build faster

Developers should get fast feedback from the CI server on the last integration build in either case – no matter if the build failed or passed. If the build takes too long, feedback is given at a time where a developer could be working on another task already. Google provided new Android emulators with every new Android version they presented. Latest emulators running on powerful machines respond nearly as fast as physical devices. This fact enabled developers the possibility to run Catroid's UI test suite simultaneously on multiple emulators instead of just on a single physical device. Due to the breakdown of the test package org.catrobat.catroid.uitest, which contains all of Catroid's UI tests, into several sub packages it is easy to execute package A on emulator A, package B on emulator B, and so on. After the tests have finished the CI server collects the testresults from all emulators and physical devices involved and merges them into a single report. With this change the runtime of an integration build was cut down from over two hours to under 30 minutes.

Unfortunately it was not possible to run the Emma[1] code coverage tool any more and the Catroid team lost the possibility to observe the coverage trend (see Figure 4.2 for the coverage trend up to the time when the test suite was distributed on multiple emulators). Code coverage is a metric in software testing, presenting the number of lines in a source code, which were actually tested.

[1]http://emma.sourceforge.net/; last visit: 2014-09-22

The split-up of the test cases solved the memory problems during the test runs, because instead of running over two hours straight, each emulator only needs about 30 minutes to test. Theoretically, on a very powerful test machine, a large number of emulators could be started simultaneously. However, the observations showed that running more than five emulators on the same machine at the same time does not work well, because the ADB connection is lost randomly and the testrun is aborted. To increase the number of emulators on a machine to a maximum, there is the possibility to set up different virtual machines on a physical one, and just allow two or three emulators on each VM. Since each instance of the ADB is running in its own environment, the random crashes should decrease significant. The downside is the additional overhead of memory and CPU power needed on the physical server to run all virtual machines, but it should be possible to run many more than five emulators simultaneously on a suitable machine.

4.4 Faster Code Acceptance

When a Catroid developer wants to commit to the mainline, the changes have to be reviewed by another team member. To provide faster feedback, the team makes use of the Github pullrequests. To support the accepting developer, the team agreed to install a so called *pullrequest test*: on every new pullrequest, a rudimentary testrun should be executed, and should print the results directly to the Github pullrequest. A tool, which allows such behavior, is Jently[2] . If a new pullrequest is created, a testrun including building the .apk file, running all unit tests and checking the source code with static analysis tools is executed. After this testrun, the pullrequest is modified so everybody can see, if the test was successful or there were any errors. Another useful feature is, that even open pullrequests get tested if

[2]https://github.com/vaneyckt/Jently; last visit: 2014-05-25

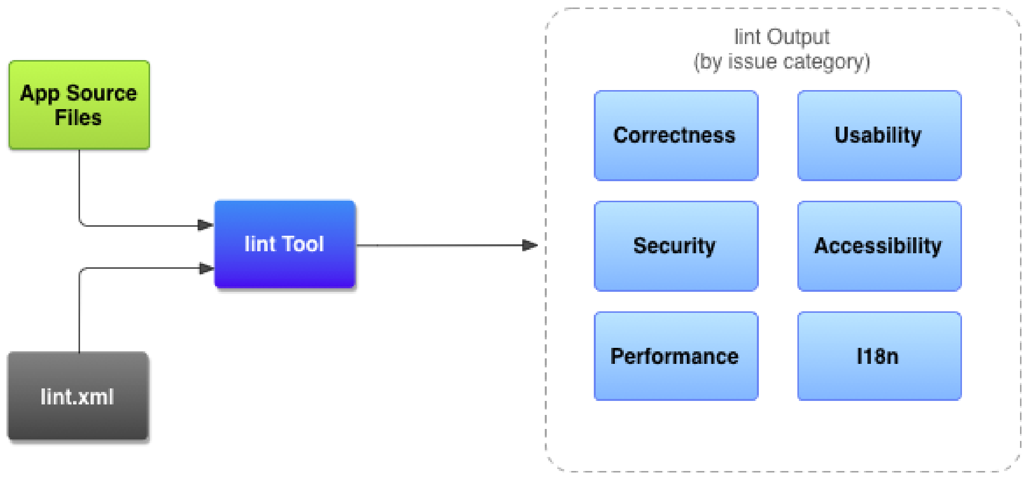


Figure 4.4: Android Lint [10]

there were new commits. Thus, any reported errors can be easily fixed until the pullrequest test is successful and the code is ready for review.

4.5 Include Code Analysis Tools in Build Process

Since Android development is done in Java, it is possible to check the code with all static code analysis tools that support Java. While testing is a dynamic process and checks the software to test the functionality, code inspection analyzes the source code with predefined rule sets. A very important aspect is the fact that by just inspecting the code, high quality software is not achieved. For an increased code quality, developers have to look at the reports and fix reported problems [13, pp. 164-165].

4.5.1 Lint for Android

The Android platform has its own static code analysis tool, called Lint [10] - it scans the project's source files for bugs and both code and layout optimization Figure 4.4. My task was to integrate lint into the Catroid project. Since some of the checks do not make sense for our app, we use a slightly modified configuration file which excludes some checks. The main goal is to have no lint warnings at all for all important checks.

Lint is running on a regular basis on our CI server, but can be also executed manually on the developer's machine. The important aspect is, that every single warning excluding the ignored ones should be reported, so the developers can react immediately and fix the potential problems.

When running lint for the first time on Catroid, the tool reported many warnings and errors. In order to be able to set the warning threshold to zero on the CI server, the following workflow was used.

- First of all I disabled all checks which reported errors and warnings. Thus running lint reported 0 problems.
- In the next step, just a single group of additional checks was activated and lint was executed again
- Now all the warnings and errors for these specific checks were fixed and the code was committed
- Returning to the second step and activate new checks

The advantage of this approach was that each commit reported no warnings and errors, because the source code was updated as well as the lint configuration file. Right after the first commit, lint checks were activated on the CI server. Since it is possible to execute lint with the same configuration file on the developer machine there were hardly any problems introduced to the mainline.

Listing 4.1: Catroid lint options

```
lintOptions {
  lintConfig file('config/lint.xml')
  ignore 'ContentDescription', 'InvalidPackage',
    'ValidFragment', 'GradleDependency',
    'ClickableViewAccessibility', 'UnusedAttribute',
    'CommitPrefEdits', 'OldTargetApi'
  textReport true
  xmlReport true
  htmlReport false
  xmlOutput file("build/reports/lint.xml")
}
```

Listing 4.2: Catroid lint.xml file

```
<lint>
...
<issue id="MissingTranslation" severity="ignore">
    <ignore path="catroid/res/values-ko/strings.xml" />
    <ignore path="catroid/res/values-pt/strings.xml" />
    <ignore path="catroid/res/values-ro/strings.xml" />
    <ignore path="catroid/res/values-ru/strings.xml" />
    <ignore path="catroid/res/values-nl/strings.xml" />
  </issue>
...
</lint>
```

Listing 4.1 shows the lint options that are used in Catroid's Gradle build file. All warnings which should be ignored for the whole source code, are defined here. Additional ignores for specific locations are set in the lint.xml file (see Listing 4.2 for an excerpt of the Lint configuration file). These

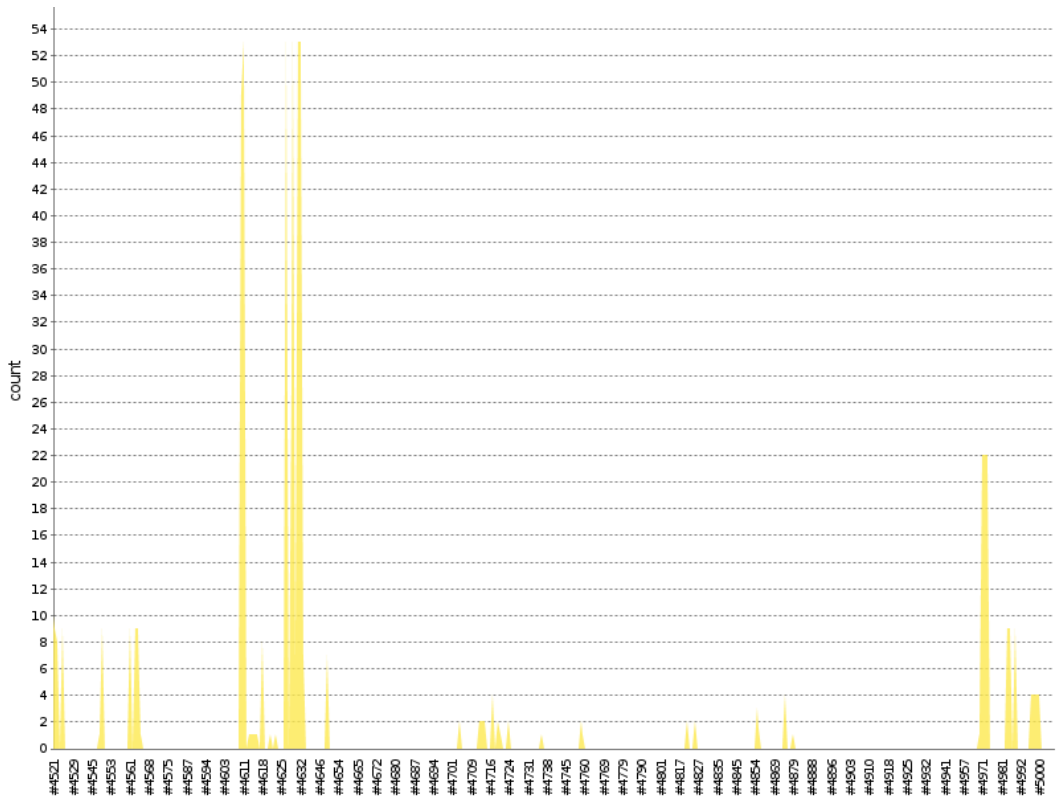


Figure 4.5: Catroid Checkstyle analyzation for pullrequest tests

additional ignores for example exclude the *Missing Translation* warning for all languages beside German and English, since in the Catroid project, these languages are serviced by external members and might not be up to date. The output of the lint check is a console output and an xml file, that is not only used by the CI server to publish the results, but also by developers on their local machine to inspect possible warnings and errors.

4.5.2 Checkstyle

Another static code analysis tool I integrated into Catroid's CI process is Checkstyle[3] . With this popular tool it is possible to test source code to follow a certain coding style. The integration process was done in a similar way as with Lint: in the beginning, only a few checks were activated in a separate configuration file called *checkstyle.xml*. After running Checkstyle on Catroid's source code, all reported warnings were fixed, the changes were committed to the VCS and the CI server used the configuration file to run Checkstyle on every single push to the mainline to report any violations of the coding style. Over the time, more and more checks were enabled together with the corresponding changes in the source code to fix the potential warnings.

Listing 4.3: Catroid checkstyle.xml file

```xml
<module name="Checker">
  <property name="severity" value="warning"/>
  <module name="TreeWalker">
    ...
    <module name="MemberName">
      <property name="format" value="^(([a-z]{2})|([x-z][A-Z]))[a-zA-Z0-9]*$"/>
      <message key="name.invalidPattern" value="Member name not following naming convention - Name ''{0}'' must match pattern ''{1}''."/>
    </module>
    ...
    <module name="ArrayTypeStyle"/>
  </module>
</module>
```

[3] http://checkstyle.sourceforge.net/; last visit: 2014-09-15

In Listing 4.3 there are two sample checks shown that are activated for the Catroid project. The *MemberName* check ensures that class members follow a certain naming convention. With a regular expression it is easy to adapt checks to nearly any coding style. The second check *ArrayTypeStyle* guarantees, that only Java array style is accepted, while C style arrays throw a warning.

While in the master build nearly no Checkstyle errors are reported, because most developers execute the checks on their development machines, the pullrequest check reported some warnings. If the check was not executed, the violations of the coding style might have been committed to the mainline, because it is not guaranteed that the member accepting the code would have seen the problems. Figure 4.5 shows a long term observation of 500 pullrequest tests and how many violations would have been merged to the mainline, if someone had accepted the code without looking at the results and fixing the violations.

4.5.3 PMD

The third tool that I added to Catroid is Pmd[4] . While there are some overlapping checks with Checkstyle, Pmd also checks for example for unused variables, empty blocks, large classes and code complexity. In the beginning, also with this tool I chose the approach of activating only a few checks, fix the potential errors and activate more checks. Initially, the warning threshold on the CI server was set to match exactly the warnings in the current build to prevent additional violations in the future. But as soon as all the warnings were fixed, the threshold was set to zero, and from that time on, no warnings were reported on the mainline build (see Figure 4.6),

[4]http://pmd.sourceforge.net/; last visit: 2014-09-16

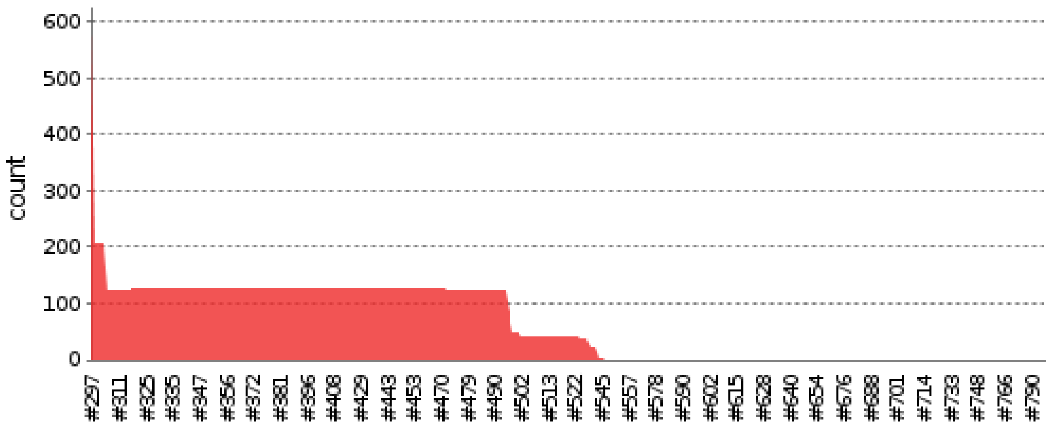


Figure 4.6: Catroid Pmd long term analysis

since the violations are reported on the pullrequest test as well as a last barrier.

4.5.4 Findbugs and CPD

While Lint, Checkstyle and Pmd fail Catroid's CI build in case of any errors, Pmd-Cpd[5] and Findbugs[6] are executed only as long term analysis. Copy/Paste Detector (CPD) is an extension for Pmd to detect duplicate code in a project. Figure 4.7 shows a long term graph of duplicate code in the Catroid project, dividing the issues found into warnings with high priority (red), normal priority (yellow) and low priority (blue). It would be good to set a certain threshold to fail the build in case of too many warnings here as well, but is very difficult to find the appropriate threshold value.

Findbugs works slightly different than the other tools, because it needs compiled Java .class files to execute checks for common Java programming

[5]http://pmd.sourceforge.net/pmd-4.3.0/cpd.html; last visit: 2014-11-01
[6]http://findbugs.sourceforge.net/; last visit: 2014-09-18

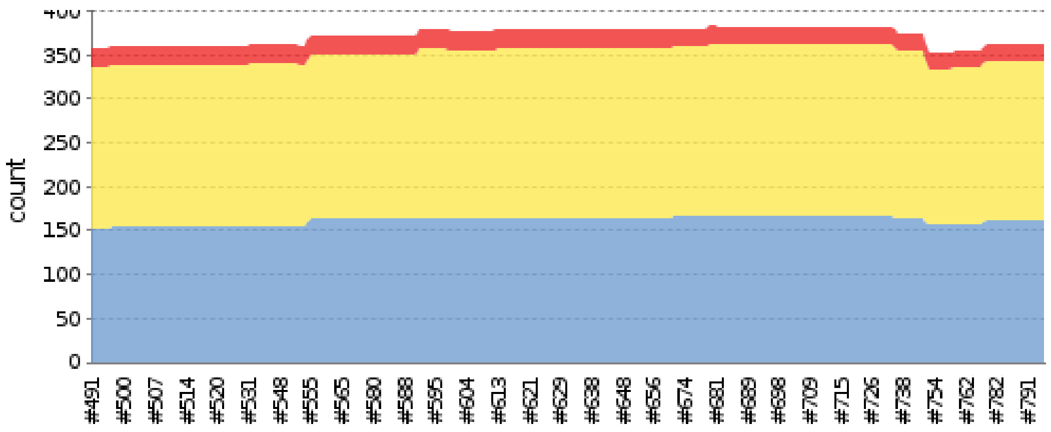


Figure 4.7: Catroid Pmd CPD long term analysis

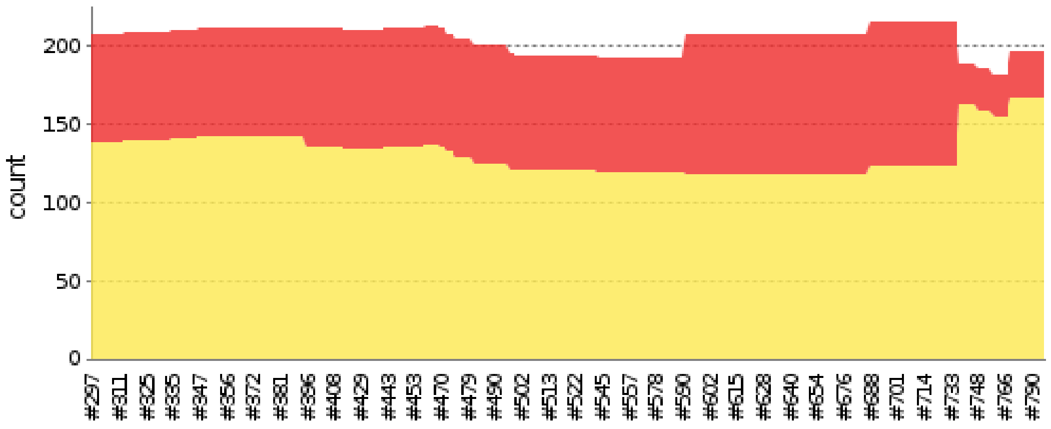


Figure 4.8: Catroid Findbugs long term analysis

errors. Figure 4.8 shows Catroid's Findbugs long term graph with high (red) and normal (yellow) priority warnings.

It can be observed easily, that warnings from both of these tools did not drop significantly over a few hundred builds. To increase Catroid's internal software quality even more, CPD and Findbugs should be included in the strict CI process as well.

4.6 Improve Feedback Mechanism

At the beginning of this thesis, the feedback mechanism was only weakly defined. While developers were looking at the results of their custom branch tests frequently, a failing mainline build was not be recognized by developers. In case of a broken build an e-mail was sent to the Jenkins administrators and the web interface marked the build as red or yellow depending on the error. Jenkins comes with a huge amount of plugins which can be easily installed and integrated in different jobs. When the development team changed internal communication channels from Skype to IRC, the Jenkins server was extended with a plugin for IRC notifications. After every master build a message is posted to the internal IRC channel containing the build status and a link to build on the CI server.

```
[05:48:32] <catrobat-jenkins> Project Catroid-Multi-Job
  build #403: SUCCESS in 40 min:
  https://jenkins.catrob.at/job/Catroid-Multi-Job/403/
```

This way developers are notified of failing builds immediately.

4.7 One Iteration in Catroid's Development Cycle

In this section a whole iteration from assigning to a ticket up to getting feedback from the CI server is discussed.

- Developer or pair of developers assigns to an issue on Github or takes a ticket from the agile board in the team room with the highest priority available
- Developer has to pull the master branch from the VCS and creates a local feature branch from the current master
- Development begins – a test is written first and executed → test fails; afterwards the corresponding code is implemented to make the new testcase pass. After the test is passing, some code refactoring can take place. The next step is to run all unit tests locally. If this test run is successful, the changes can be committed to the local branch (TDD cycle ends here)
- The master branch is pulled again; the developer merges the local feature branch with master branch or rebases the feature branch onto master → depending on new commits in the mainline, merge conflicts have to be resolved within this step. However, the difference should not be significant due to the short time span between starting development and pulling again
- The new local branch is pushed to the remote VCS
- Custom branch test is started for the feature branch; during this step a whole integration build is performed – this includes running all tests and code analysis
- If the build is successful, a pullrequest is written and the link to the successful build is provided for the reviewer. In case of any problems with the custom branch test, development starts again with the TDD cycle

- If the pullrequest is finally merged, the CI server recognizes the changes in the mainline, triggers a master build and notifies developers with the IRC bot and updated web interface.

This development cycle has one downside – if every developer is following this workflow, the second testrun on the CI server is obsolete, because the same revision is tested twice. In a CI environment with limited resources such a scenario should be avoided. A possible solution to this problem would be to check the latest test runs in the beginning of the automatically triggered master build. If the same revision is found, and the build status is successful, the build script should only copy the artifacts needed for long term analysis instead of performing a full build. This way, the build time for the second run decreases from 40 minutes to about 1 minute.

5 Analyzing Fowler's Continuous Integration best Practices

Martin Fowler presents ten key practices of CI in his updated article[15].

In this chapter these practices are summarized and compared with Catroid's current CI process.

5.1 Maintain a Single Source Repository

As described in 3.1 a version control system for source code management is needed for CI. Fowler [15] explains that it is essential to put everything in the VCS that is needed to build a software system. Additionally to source code at least source code for tests, install scripts, libraries, property files and IDE configuration files should be located within the repository. It should be possible to build after a check out on a new machine with just the operation system, compiler and database system installed.

The Catroid team uses Git as a VCS. Many different types of files are put in the repository:

- Source code
- Test code
- Third party libraries

- Bluetooth server code needed to compile bluetooth server for testing
- IDE configuration files for Android Studio
- Configuration files for static code analysis
- Configuration files for the Git repository
- Gradle build scripts for Android

Due to the difficile building process on Catroid's Jenkins server with distributed testing on multiple emulators simultaneously, it is not possible to run a full build on a local machine after checking out the Catroid repository. The scripts needed for Jenkins were located in a different repository for a long time. However, since the migration from Ant to Gradle it is possible to run unit tests, source tests and code analysis from both the command line and Android studio with a single command after cloning Catroid, because the needed files are checked in to the repository right now.

5.2 Automate the Build

In a CI system the whole build from compiling the sources to running tests should be fully automated. Common automation tools are Ant and Maven (for Java), MSBuild (for .NET) and Gradle (for Android). Tight coupling of a build and an IDE should be avoided. While it is okay for developers to use IDE build tools on their own machines for building and testing, it is essential that on the CI server it is possible to build the whole project without using a IDE [15].

As far as Catroid's build process is concerned, the advises from above are correct. The build on the CI server is fully automated, while most developers use IDEs to build and test locally.

5.3 Make Your Build Self-Testing

Compilation of source code is not enough to be referred as a build. Automated developer tests are needed to ensure product quality. An automated build should fail if just a single test is failing [15].

Within Catroid tests are not only used to ensure that there are no bugs, but also act as documentation. To ensure a good code coverage, Catroid uses the TDD approach. By applying TDD the CI process benefits from the presence of the always increasing number of tests. Currently two categories of tests are executed during CI process:

- Unit tests
- Functional tests

Unfortunately it is nearly impossible to make the build self testing on the developer's local machine. Without splitting the testsuite to five different emulators the whole testrun would last for more than 2 hours and 30 minutes. Additionally the functional tests might not work correctly on every single Android device available. Thus the Catroid team uses a development process, where only the unit tests and single UI tests are executed on the developer machines (see 4.7). To be sure, that the new UI tests would pass on the CI server, developers should aim to setup the exact same emulators, which are used on the server, and execute the new tests locally.

5.4 Everyone Commits to the Mainline Every Day

Fowler [15] states there is only one prerequisite for committing to the mainline: a successful build. Developers should aim for multiple commits to the mainline every day. Only by committing frequently, it is possible to fix problems quickly because the conflicts must have occurred since the latest commit. In order to achieve this goal it is essential to break down tasks in smaller chunks which do not take more than a couple of hours to complete.

This particular practice is not well educated within Catroid development. Developers are not able to commit multiple times a day to the mainline and code remains unmerged in feature branches for months. There are several reasons for this dilemma:

- Developers are mainly students
- Tasks are too large
- Complicated acceptance process
- Long build duration

Most of the developers are students and they do not work eight hours straight on Catroid tasks every single day. However, even with very short tasks it is impossible to get the code accepted within a day. A bugfix including tests is potentially fixed within twenty minutes. After having pushed the feature branch to the remote repository the developer has to start a full testrun for this branch. This testrun takes approximately 45 minutes to complete, but the feedback time can be much longer in case there are several test runs in the waiting queue. When the build is successful a Github pullrequest is written and a senior developer has to accept the code. In some cases there are over twenty open pullrequests and due to the fact, that senior developers are students with limited time also, it can take up to a week until this bugfix is merged. Meanwhile other tasks with many changes could be merged to the mainline and could possibly make the bugfix unmergeable because of compilation errors or failing test cases. This is one of the key practices that should be improved, but it can only be done with a complete restructuring of the development process.

5.5 Every Commit Should Build the Mainline on an Integration Machine

The mainline, which is sometimes referred to the HEAD of development, should always stay in a stable state. Since every single commit to this mainline should trigger a CI build there are several builds a day. However, even with developers running private builds before committing, these builds fail from time to time because of developers forgetting to commit all files or running no private builds at all. A broken mainline build is not a disaster, but is has to been fixed right away. Developers should monitor the CI build after their commit and should fix a potential problem immediately [15]

In Catroid's development process a CI build is triggered right after a commit to the master branch. Due to the mandatory execution of a build on the CI server before committing to the mainline the number of failed builds caused by developers is very low. Possible situations for failed builds caused by developers are unintended commits to the master branch and a long period between successful custom branch test and merge, which could cause integration problems in case of an updated master branch.

5.6 Keep The Build Fast

Fowler [15] describes the problem that a build has to be fast. Otherwise it is impossible for the developers to get feedback right on time. They might work on other tasks in the meantime and in case of a broken build, it could take more time to fix the build compared to if they got faster feedback [15]. Duvall also advises to use some kind of build pipeline - in his opinion a build should consist of at least two stages [13, pp. 92,93]:

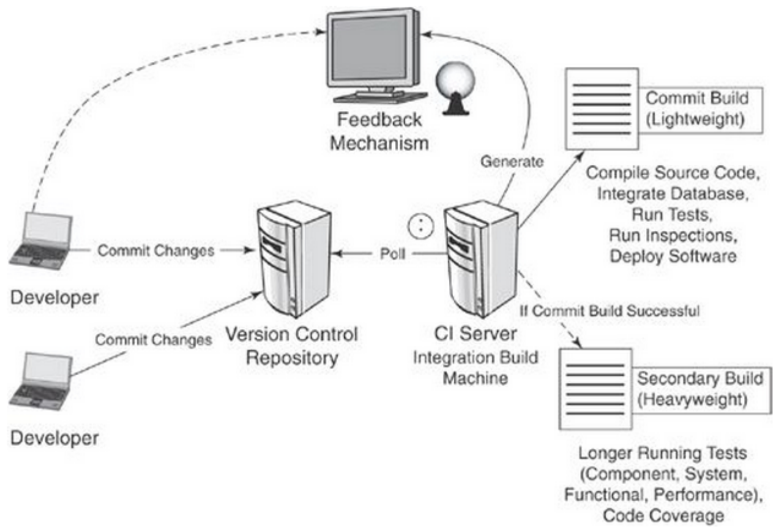

Feedback Mechanism
Commit Build (Lightweight)
Generate
Compile Source Code, Integrate Database, Run Tests, Run Inspections, Deploy Software
Commit Changes
Poll
Developer
Commit Changes
Version Control Repository
CI Server Integration Build Machine
If Commit Build Successful
Secondary Build (Heavyweight)
Developer
Longer Running Tests (Component, System, Functional, Performance), Code Coverage

Figure 5.1: Staged Build [13, p. 93]

- lightweight commit build
- heavyweight secondary build

Figure 5.1 demonstrates this advice. Every new commit triggers a build which compiles the code, runs unit tests, runs code analysis and deploys the product. If the commit build fails, developers would get feedback within minutes and could immediately start to fix the broken build. In case of a successful commit build, a secondary build is started, which executes longer running functional tests [13].

Within Catroid, this practice is developed worst. The build time is far away from ten minutes - it was about 30 minutes, when CI was introduced and increased to over two hours. With parallelization of the testsuite onto multiple emulators the build time was minimized to about 40 minutes - however, the developers are annoyed of the waiting time, because many new builds would be queued up due to the fact that only one job with emulators can run at the same time. Sometimes developers have to wait for over two hours until their job is starting, and get feedback after three hours.

5.7 Test in a Clone of the Production Environment

Testing should reveal problems of the system in similar conditions as in production. Fowler [15] describes a rule that it is very important to test in an environment, which is as much as possible a clone of the production environment.

Catroid's automated tests are executed on both real hardware devices and Android emulators. However, there is a huge number of Android devices available and it is impossible to test an application on all of these devices. While it is not a big deal to test unit tests on different devices, it is very hard to write automated functional tests running stable on different devices. By using Jenkins it is possible to create so called matrix-builds which execute tests in different environments. However, as long as the UI tests do not run at a stable state on the existing emulators and test devices, it makes no sense to enlarge the scope of devices to test on.

5.8 Make it Easy for Anyone to get the latest Executable

According to Fowler [15] it is essential that anyone within a project can get the latest executable. On Catroid's CI server Jenkins all executables from mainline builds are archived and can be downloaded. Additionally the latest successful build artifacts are copied to a fileserver which is linked on the developer website.

5.9 Everyone can see what's happening

Every project member should know the health status of the CI build. Within Catroid, this information gathering is mostly done via the Jenkins' web interface. Due to this fact, this practice is working well for custom branch tests and other jobs, that are triggered manually, because the developer wants feedback for the build. However, most developers do not recognize, if the mainline build fails. After a pullrequest was accepted, nobody is waiting for the integration build, and so it is possible, that a build remains broken for several days. In order to inform developers of the health status of the master build, an IRC notification was installed to post the result of any integration build to the developer's channel. Additionally, Jently was installed to check open pullrequests automatically and update the Github web interface accordingly. Since Catroid's testsuite takes a long time, only a small subset of tests and inspections is run, but this build verifies, that the code from a pullrequest – both, from external developers and internal ones – can be built, the unit tests are running and the code is following the team's coding conventions.

5.10 Automate Deployment

This practice is well developed within Catroid. During a CI build, the generated .apk files are deployed on both, real devices and emulators. Furthermore, a nightly build version is deployed on a fileserver and everyone interested in the latest features and bugfixes can download the .apk file and test it on any Android device. If a certain revision is proposed for a Google Playstore release, on top of the CI build, several steps have to be done by hand:

- sign .apk file with release key
- upload .apk file to Google Play
- write change log to inform users of the changes
- make new meaningful screenshots and update the Google Play site

While the signing can be possibly performed by the CI server, uploading the .apk file and writing the change log for the Google Playstore has to be conducted manually.

6 Future work

6.1 Categorize Automated Tests

The tests in the Catroid project are not perfect from several points of view. First of all, there is no real distinction between different kind of tests. Duvall explains four different test categories [13, pp. 132f]

- Unit tests, which test the behavior of single classes without relying on outside dependencies like databases or internet connection
- Component tests, which have more dependencies than unit tests and thus running a bit slower
- System tests require a fully installed system and test the whole system with all components
- Functional tests, which test the system from a client point of view - they are also called UI and acceptance tests

In Catroid, there exist two major test packages: unit tests and UI tests. However, even the tests in the unit test package are not real unit tests. Some of the tests need an active internet connection, otherwise they would fail. Additionally it is not possible to execute the unit tests on an Android emulator, since some tests require real hardware sensors. These problems limit the capability of running quick private builds. Thus developers need to use the CI server to execute all the unit tests, which takes much more time than running them locally.

In the UI test package, there is a conglomeration of component, system and functional tests. Rogers [25] states that such a scenario has to be avoided, since the tests became less and less maintainable. For the functional tests, Catroid's developers use a tool called Robotium[1] , which provides the functionality to test Android applications like a client would do. Unfortunately, some of these tests are not correctly written in the Catroid project, because inside of real UI tests, there are checks for several internal states of the Catroid application. This behavior is the reason, why with only some changes in the Catroid app, which are not related to the user interface, many UI tests have to be refactored as well and the time spent on maintaining the test suite is too large.

In order to be able to run tests at different levels, the Catroid team should at least split up the test packages into:

1. Java unit tests, which do not require an Android .apk file to be created
2. Android unit tests, which test all the classes that need components of the Android system
3. System tests, which test the core of the Catroid application
4. UI tests, which test the app only from a client's perspective and only need to be changed if the user interface is changing

6.2 Make the Build faster

In order to reduce the runtime of the CI build, there are several options:

- Increasing the number of build servers and try to run the tests on even more emulators

[1]http://code.google.com/p/robotium/; last visit: 2014-04-14

- Refactoring of the tests by moving test cases from slow UI tests to fast unit tests as much as possible
- Using Robolectric[2] to run Android related tests outside of an Emulator or physical device directly inside the Java VM on the developer machine
- Setting up the staged build described in 5.6

While the Catroid development team has agreed not using the staged build approach, because errors could be introduced to the mainline more easily and they probably would not be fixed in a short enough time span, all other three options could make the CI build significantly faster.

6.3 Faster Development Iterations

The development cycle for Catroid has become quite complex over the time. Unfortunately, the time span between starting a story and the final merge into the mainline is very large, even for very small bugfixes. Another problem is the complexity of single stories – CI works best, if the stories are small, and multiple commits are performed every single day. However, in the Catroid project, some tickets are really large respective the hours it takes to finish them and it is possible that some stories are open for months. Even if the developer, who is responsible for this feature branch, is updating the branch frequently with the mainline, other developers would not recognize the changes and merge conflicts are inevitable at the time the feature branch is merged to the master branch.

[2]http://robolectric.org/; last visit: 2014-04-14

6.4 Integrate additional test methods

As mentioned earlier, Catroid's test suite mostly consists of unit tests and UI tests. However, there are other methods, which could ensure the quality of the app. One possibility is to use a tool called *monkey* on the CI server. This is a program which sends random inputs to the Android app and is working on the emulator as well as on a physical device. Thus the application is stress tested and potential crashes can be reproduced [12].

Milano describes the so called monkey theorem. This theory defines that a monkey typing random characters on a keyboard for an infinite amount of time will type any given text [22, p. 142].

The monkey is already running on Catroid's CI server, but is not yet integrated into the continuous integration process. It is executed as nightly build, because adding it to the master build would extend the build time even more. During these nightly builds, the monkey is running 10 000 random events four times in a row, and possible crashes are reported to the CI server's web interface. However, as already mentioned earlier, just reporting the errors is not enough – developers have to look at the results and investigate the potential problems in the software. This is currently not done, but in the future, the monkey should be executed on different Android devices and emulators, and the results should be discussed in the weekly development meeting.

7 Conclusion

What problems emerged during development using continuous integration?

Over the time, between introducing continuous integration in Catroid's development process and some iterations later, the main problem was the long build time of a CI build. Android projects might have a difficult time to get to the proposed *ten minute build*, because the compilation takes some time, it is hard to write pure unit tests and additionally the functional tests are slow on both the emulator and any physical device. By running a sub set of all the tests on multiple emulators simultaneously, the team was able to reduce the runtime, but without any maintenance, the build time kept increasing by and by again.

Furthermore the acceptance for continuous integration amongst the team members is not high enough. Not all developers recognize the benefits of the CI server, but instead complain about the additional overhead of running the builds on their branches on the server and having to fix all the warnings and errors reported.

What key practices of CI have helped to make the whole development process more stable?

A key practice in Catroid's CI process, which is also described by Lacoste [18], is the use of the custom branch tests. Without the possibility to execute full test runs on specific branches, the development team would have a hard

Figure 7.1: Ambient Orb

time to fix all the failing tests that would arise when building the mainline without testing the feature branch before.

As far as code quality is concerned Catroid's CI server now runs multiple tools on every build:

- Lint for Android to detect any possible bugs in layouts and Android related source code
- Checkstyle to force developers to adopt to naming conventions
- Pmd to find developer mistakes
- Findbugs as long term analysis tool to detect bad Java development

Furthermore every single pullrequest on Github is analyzed automatically including compilation, unit tests and source code analysis. This way a senior member, who is responsible to merge this new feature or bugfix to the mainline, receives quick feedback on possible integration errors.

What can be done to optimize CI for Catroid?

Generally the Catroid development team adopts to many continuous integration practices described by Duvall and Fowler. The most important optimization for Catroid's CI process is to reduce the runtime significantly and to prevent the tests from failing randomly. Only with fast and stable feedback from the CI server, developers can get the most out of continuous integration – otherwise too much time is spent on investigating the problems with the latest builds.

Another possible optimization is the enhancement of the CI server's feedback. Currently developers have to check the IRC channel or the Jenkins' web interface to get any feedback from the CI server. For all co-located team members it could be a good idea to setup an extreme feedback device like an Ambient Orb[1] (see Figure 7.1), notifying all developers in the team room of the latest build status by changing the light accordingly. That way a broken build would be recognized much quicker and the time it takes to fix this build might go down. Additionally, as described by McGregor [20], longer unstable states of the mainline could decrease, because developers might start to work on the problem more quickly if a red light is indicating an error instead of an IRC message.

Furthermore it should be a goal to reintegrate EMMA code coverage once again. Only with a good code coverage tool it is possible to observe potential missing tests, which could have an impact on the software quality.

[1]http://ambientdevices.myshopify.com/products/energy-orb; last visit: 2014-11-29

Bibliography

[1] F. A. Abdul and M. C. S. Fhang. “Implementing Continuous Integration Towards Rapid Application Development.” In: *2012 International Conference on Innovation Management and Technology Research (ICIMTR)*. Malacca, Malaysia: IEEE, May 2012, pp. 118–123. DOI: `10.1109/ICIMTR.2012.6236372`.

[2] K. Beck. *Extreme Programming Explained. Embrace Change*. 1st ed. Addison Wesley, 1999. ISBN: 0201616416.

[3] K. Beck. *Extreme Programming Explained. Embrace Change*. 2nd ed. Addison Wesley, Pearson Education, 2004. ISBN: 9870321278654.

[4] K. Beck. *Test-Driven Development. By Example*. Addison Wesley, Pearson Education, 2003. ISBN: 9780321146533.

[5] J. Bowyer and J. Hughes. “Assessing Undergraduate Experience of Continuous Integration and Test-driven Development.” In: *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. Shanghai, China: ACM, May 2006, pp. 691–694. DOI: `10.1145/1134285.1134393`.

[6] S. Chacon. *Pro Git*. Apress, 2009. ISBN: 9781430218333.

[7] L. Crispin and J. Gregory. *Agile Testing. A Practical Guide for Testers and Agile Teams*. Addison Wesley, Pearson Education, 2009. ISBN: 9780321534460.

[8] Developer-Android. *App Fundamentals*. URL: http://developer.android.com/guide/components/fundamentals.html (visited on 03/18/2014).

[9] Developer-Android. *Building Android Applications*. URL: http://developer.android.com/tools/building/index.html (visited on 03/18/2014).

[10] Developer-Android. *Improving your code with Lint*. URL: http://developer.android.com/tools/debugging/improving-w-lint.html (visited on 08/13/2014).

[11] Developer-Android. *Sensors Overview*. URL: http://developer.android.com/guide/topics/sensors/sensors_overview.html (visited on 03/18/2014).

[12] Developer-Android. *UI/Application Exerciser Monkey*. URL: http://developer.android.com/tools/help/monkey.html (visited on 09/25/2014).

[13] P. M. Duvall. *Continuous Integration. Improving Software Quality and Reducing Risks*. Addison Wesley, Pearson Education, 2007. ISBN: 98703213336385.

[14] M. Fowler. *Continuous Integration*. 2000. URL: http://martinfowler.com/articles/originalContinuousIntegration.html (visited on 02/24/2014).

[15] M. Fowler. *Continuous Integration*. 2006. URL: http://www.martinfowler.com/articles/continuousIntegration.html (visited on 02/24/2014).

[16] S. Freeman and N. Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison Wesley, Pearson Education, 2010. ISBN: 9780321503626.

[17] J. Hembrink and P.-G. Stenberg. “Continuous Integration with Jenkins.” In: Faculty of Engineering, Lund University (LTH), Sweden, Mar. 2013.

[18] F. J. Lacoste. “Killing the Gatekeeper: Introducing a Continuous Integration System.” In: *Agile Conference (AGILE '09)*. Chicago, USA: IEEE Computer Society, Aug. 2009, pp. 387–392. DOI: `10.1109/AGILE.2009.35`.

[19] J. Loeliger. *Version Control with Git. Powerful Techniques for Centralized and Distributed Project Management*. O'Reilly Media, 2009. ISBN: 9780596520120.

[20] G. McGregor. “A 30 Minute Project Makeover Using Continuous Integration.” In: Verilab, Inc. – Austin, Texas, USA, 2012–02.

[21] R. Meier. *Professional Android 4 Application Development*. John Wiley & Sons, 2012. ISBN: 9781118102275.

[22] D. T. Milano. *Android Application Testing Guide*. Packt Publishing, 2011. ISBN: 9781849513500.

[23] A. Miller. “A Hundred Days of Continuous Integration.” In: *Agile 2008 Conference (AGILE '08)*. Toronto, Canada: IEEE Computer Society, Aug. 2008, pp. 289–293. DOI: `10.1109/Agile.2008.8`.

[24] B. O'Sullivan. *Mercurial. The Definite Guide*. O'Reilly Media, 2009. ISBN: 9780596800673.

[25] R. O. Rogers. “Scaling Continuous Integration.” In: *Extreme Programming and Agile Processes in Software Engineering, Proceedings 5th International Conference (XP 2004), LNCS 3092*. Garmisch-Partenkirchen, Germany: Springer Verlag Berlin-Heidelberg, June 2004, pp. 68–76. ISBN: 9783540221371.

[26] W. Slany. "A Mobile Visual Programming System for Android Smartphones and Tablets." In: *Proceedings of the 2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Innsbruck, Austria: IEEE, Sept. 30–Oct. 4, 2012, pp. 265–266. DOI: `10.1109/VLHCC.2012.6344546`.

[27] W. Slany. "Catroid: A Mobile Visual Programming System for Children." In: *Proceedings of the 11th International Conference on Interaction Design and Children (IDC '12)*. Bremen, Germany: ACM, June 2012, pp. 300–303. DOI: `10.1145/2307096.2307151`.

[28] G. de Souza Pereira Moreira et al. "Software Product Measurement and Analysis in a Continuous Integration Environment." In: *2010 Seventh International Conference on Information Technology: New Generations (ITNG)*. Las Vegas, Nevada, USA: IEEE, Apr. 2010, pp. 1177–1182. DOI: `10.1109/ITNG.2010.85`.

[29] S. Stolberg. "Enabling Agile Testing Through Continuous Integration." In: *Agile Conference (AGILE '09)*. Chicago, USA: IEEE Computer Society, Aug. 2009, pp. 369–374. DOI: `10.1109/AGILE.2009.16`.

[30] B. Vodde. "Measuring Continuous Integration Capability." In: *CrossTalk: The Journal of Defense Software Engineering. Volume 21, Number 5*. Hill AFB, Utah, USA: USAF Software Technology Support Center (STSC), May 2008, pp. 22–25.

[31] S. Wiest. *Continuous Integration mit Hudson*. dpunkt.verlag, 2011. ISBN: 9783898646901.

[32] M. Zechner. *Beginning Android Games*. Apress, 2011. ISBN: 9781430230427.