Introduction of a Continuous Integration Process in an Open Source Project

Master's Thesis

Institute of Software Technology
Graz University of Technology

Daniel Burtscher
daniel.burtscher@student.tugraz.at

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

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

EIDESSTATTLICHE ERKLÄRUNG

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

Graz, am ……. ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

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

……. ……..
date (signature)

Abstract

The merging of individually developed software components is a common problem in a software development process. This so called software integration is traditionally done at the end of the project or at the end of a project milestone which often leads to all kinds of problems and project delays.

This thesis deals with an alternative method of software integration which solves the integration problem by performing the integration process frequently. This so called continuous integration (CI) process is done more often, resulting in smaller integration steps. Instead of waiting till the end of the project and integrating the whole component at once, the integration is done frequently and only small pieces of new code are added to the mainline of the project. Doing the integration in small pieces makes it easier to solve conflicts and to find errors when merging new components. The goal of continuous integration is to make software integration a nonevent and give every developer the ability to integrate new code several times a day.

The second part of this thesis deals with the practical introduction of a continuous integration process in an open source project. CI is introduced in a project called Catroid. Catroid is an Android application designed for kids and enables them to learn programming skills in an easy way by using a visual programming language.

Introducing CI in the open source project Catroid comes with several benefits especially as the software is developed by a large development team. A defined process of how to add code to the mainline of the project is needed to ensure that the code of the application is tested and of high quality.

A method is introduced which allows the measurement of the CI capability of a project and allows a quick overview of the CI process. It is possible to identify problems in the CI process or to share practices between different projects. It shows opportunities where the CI process in a project can be made even better.

Zusammenfassung

Das Zusammenführen von individuell entwickelten Softwarekomponenten ist ein bekanntes Problem in einem Softwareentwicklungsprozess. Diese so genannte Softwareintegration wird üblicherweise am Ende eines Projektes oder am Ende eines Meilensteins eines Projektes durchgefürt, was oft zu allen Arten von Problemen und Projektverzögerungen führt.

Diese Arbeit befasst sich mit einer alternativen Methode der Softwareintegration, die das Problem löst indem der Integrationsprozess häufig durchgeführt wird. Dieser so genannte Continuous Integration (CI) Prozess führt zu kleineren Integrationsschritten. Anstatt bis zum Ende des Projektes zu warten und dann die ganze Softwarekomponente auf einmal zu integrieren, wird häufig und nur in kleinen Schritten neuer Code zum Projekt hinzugefügt. Diese Integration in kleinen Schritten macht es einfacher Konflikte zu lösen und Fehler zu finden wenn verschiedene Teile der Software zusammengeführt werden. Das Ziel von Continuous Integration ist die Softwareintegration sehr einfach zu machen und jedem Entwickler die Möglichkeit zu geben, neuen Code mehrmals täglich zu integrieren.

Der zweite Teil der Arbeit beschäftigt sich mit einer Einführung eines CI Prozesses in einem Open Source Projekt namens Catroid. Catroid ist eine Applikation, die für Kinder entwickelt wird und es ermöglicht Programmierkenntnisse in einfacher Art und Weise mittels einer Visuellen Programmiersprache zu erlernen.

Die Einführung von CI im Open Source Projekt Catroid hat viele Vorteile, besonders weil die Software von einem großen Team entwickelt wird. Ein definierter Prozess der beschreibt wie neuer Code zum Projekt hinzugefügt wird ist erforderlich um sicherzustellen, dass neuer Code getestet und von hoher Qualität ist.

Es wird eine Methode eingeführt, die eine Messung des CI Potentials eines Projektes erlaubt und einen schnellen Überblick über einen CI Prozesses gibt. Es ist möglich Probleme in einem CI Prozess zu erkennen, Fachwissen zwischen Projekten auszutauschen oder Verbesserungsmöglichkeiten aufzuzeigen.

Acknowledgements

Thanks to all who supported me while I wrote this thesis.

I would particularly like to thank
Prof. Wolfgang Slany
Martin Burtscher
all of my reviewers and
all members of the incredible Catroid team.

Contents

Part I

Theoretical Background

Chapter 1

About Continuous Integration

1.1 Introduction

The integration process is an important part of software development. Many developers who work on different components of a software project are involved. These components are developed independently of each other. Combining these software components into a software system and determining how they work together is called integration. This work can be difficult and time consuming, because the components often do not work together as intended.

The software integration is often done at the end of a project or a project milestone. This late integration leads to various problems.

> "Waiting until the end of a project to integrate leads to all sorts of software quality problems, which are costly and often lead to project delays. CI addresses these risks faster and in smaller increments." [8]

Since the traditional integration process often leads to project delays, continuous integration (CI) deals with this problem in a completely different way.

To avoid late integration and all the problems it creates the integration is carried out after every change a developer commits to the mainline. Because developers commit their code frequently in a CI process, the integration is also carried out often. Conflicts and errors are detected sooner and can be solved at an early stage of the development process. This reduces the risk of project delays because an integrated version of the software is available after every commit to the mainline.

The whole integration process including tests is done automatically on an integration server. This allows integration after each commit from a

developer and makes software integration a nonevent [8, p. 14, p. 21].

1.2 Motivation

Continuous integration is more than just a tool which executes software tests on a test server. CI is about improving the whole development process. John Ferguson Smart describes CI as:

> "A good CI infrastructure can streamline the development process right through to deployment, help detect and fix bugs faster, provide a useful project dashboard for both developers and non-developers, and ultimately, help teams deliver more real business value to the end user. Every professional development team, no matter how small, should be practicing CI." [20, p. 1]

He points to the main features of a CI process. He talks not only about developers and the benefits of CI to them but also about non-development personnel, such as managers. They are continuously informed about the current state of the project.

Paul M. Duvall [8, p. 29] describes the value of CI at a high level. He describes five points which are discussed below.

1.2.1 Reduce Risks

In CI the software integration is a nonevent and is done many times a day. The risk of wasting time to integrate at the end of the project is reduced.

Errors are detected as soon as possible. With each commit to the mainline the developers are informed of the test results. Each developer is responsible for not breaking the build and fixing these errors as soon as possible.

After every build the developers get a feedback on the results. This feedback is not only about test results. It is also about quality measurements, test coverage and other quality related measurements. So the developers receive a continuous view of the health of the software. With automated code inspection tools this software health is measurable and can be tracked over the development time of the software.

This data is used to make better decisions and thus minimize the risk of incorrect decisions. Assumptions are reduced in the decision-making process. The software is rebuilt in a clean environment with every commit to the mainline.

The whole build process is performed on the test server using the same scripts in every build. This ensures that the build process is done the same way every time the project is built [8, p. 29f].

1.2.2 Reduce Repetitive Manual Processes

Manual tasks, that the developers must do many times a day are time consuming and there is always a risk of failure. If most of the tasks to build the project are done automatically the risk of failure is reduced and the tasks are definitely performed. It also ensures that the tasks are always performed in the same manner.

The developers do not have to think about the building process. They can focus on productive work for the project. This automation saves time and money [8, p. 30].

1.2.3 Generate Deployable Software

Generating a new deployable version at each change in the mainline is a further development of the so called nightly build were a new binary is available every night.

Producing deployable software with every build on the test server comes with several advantages. A executable binary of the software is available with every build. So an up-to-date binary is always available and can be released at any point in time.

It is ensured that the deployment process works as intended. The main advantage is that the developers are informed immediately if there are problems with the deployment process [8, p. 31].

1.2.4 Enable Better Project Visibility

The CI process includes the generation of project related data such as code quality metrics or test coverage data. As this data is produced with every build, evolution statistics of the project are available. Statistics such as the test coverage over time can be recorded. With this project related realtime data it is possible to notice trends or help make effective decisions [8, p. 31].

1.2.5 Establish Greater Confidence

As the developers know the state of the software after every commit to the mainline, the confidence in the project is increased. On every build all the related data such as test results or quality measurements is evaluated automatically and developers are informed. If problems occur they are reported immediately and developers can react accordingly. This creates the confidence that no existing functionality has been broken.

Without this mechanism the members of a team could become insecure about the state of the software. There is always the possibility that every code change breaks the build or that an existing and perfectly working piece of functionality is no longer working [8, p. 32].

1.3 What is Continuous Integration?

CI is about performing the software integration frequently instead of waiting until the end of a project. If the integration is done at the end of the project it can lead to quality problems and project delays.

Martin Fowler describes CI as follows:

> "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." [9]

The goal of continuous integration is to make the integration a nonevent. To achieve this, the integration task is done without manual interactions from the developers. From an abstract view an automated script which runs on the integration server is doing the following without the need of manual interaction:

- Check the mainline of the project repository for changes. This check is done every few minutes.
- If there are changes, pull the newest source code from the mainline of the project repository.
- Build the whole software system on a clean integration environment.
- Run automated tests to determine if the software components work together properly. Errors are detected as early as is possible.
- Give the developers feedback about the results of the build. This feedback includes at the minimum the test results but may also consist of reports about the code quality, test coverage or other code analysis reports.

As these steps are done on every change in the project repository, a new version of the software is generated on every commit to the mainline of the project repository. The next section describes how the whole CI process works.

1.3.1 A Basic Continuous Integration Process

In Figure 1.1 the parts of a CI system and how they work together is shown. The continuous integration process starts with writing a pice of code and running a local build. A local build is performed on the developer's local machine, and the tests are also executed. In the example Lisa and Tim are

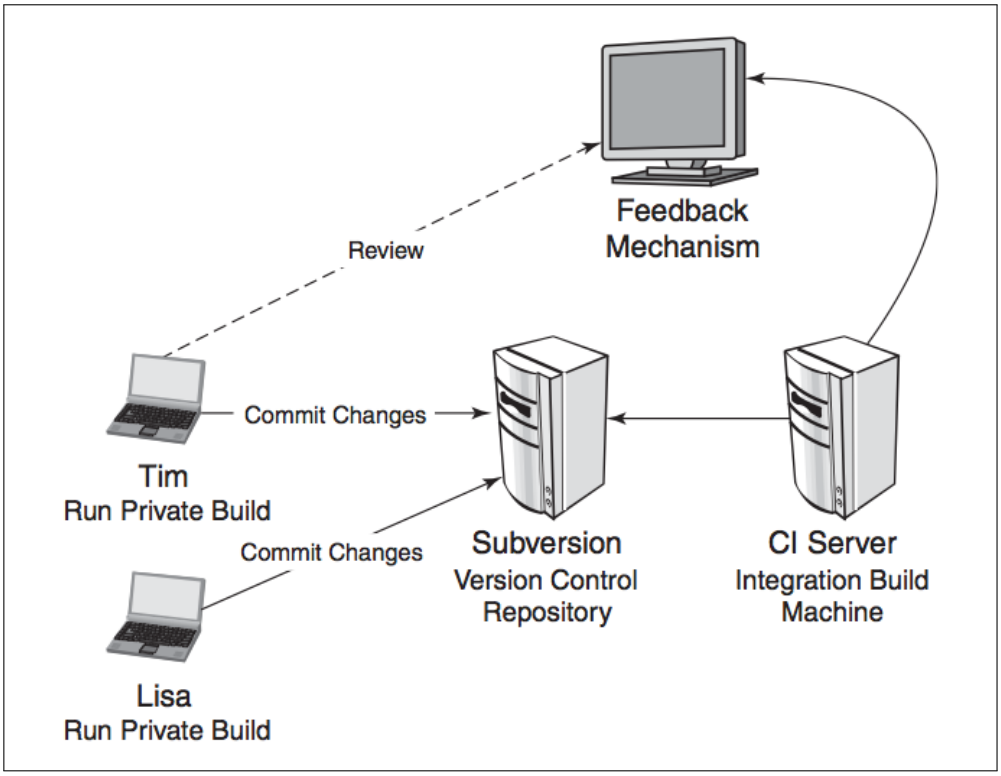


Figure 1.1: A continuous integration process [8, p. 26]

shown as the developers of the project. If the private build runs successfully, the code produced is committed to the version control repository which in this case is a Subversion repository.

The next step is that the CI server checks the version control repository for changes. If a change is detected, a build on the integration server is performed. The integration server often has a similar environment as the one the software later runs on. This means, among other things, that the same operating system is used and the same programs are installed.

It is also possible to have several integration servers with different environments. The environments can for example differ in the operating system which allows the software test to run on each of these systems. This has the advantage of developers being able to detect errors that are particular to a certain environment at an early stage and them not being spotted later on by a customer. The CI server reports these errors to the developers using the feedback mechanism.

The CI server is able to build the project by using the build script. It automates the whole build process. The tasks of this script include compiling, testing and deploying the software. It must be possible to run the whole build process with the execution of a single or at most a few scripts. With

the build script, the CI server is able to be fully automated. It can also be used outside the continuous integration environment to build the software on the local machine of the developer.

The continuous integration process ends with informing the developer of the result of his build. If the build on the integration server is finished, the developer is informed using the feedback mechanism of the CI system. This can for example be in the form of an email. If the build was successful, then the developer can continue with the next task. If the build is broken then the developer is forced to review the result of the build and fix the build as soon as possible.

A key factor of a CI system is to break the build as infrequently as possible and if it is broken for some reason, the developer who caused the break is responsible for fixing it as soon as possible. The goal is to always have a working build in the mainline of the software project [9, p. 25ff] [5, p. 29].

Chapter 2

Parts of a Continuous Integration System

In order to work properly, a CI system should consist of four parts. This chapter will describe the main parts of the CI system in detail.

Figure 2.1 shows how these parts work together. The tasks of the various parts is described below:

- The version control repository contains the whole source code of the project and all things that are required to build the software. Every developer commits the written source code to this one well defined place. The main line of the repository always contains a working build of the software.
- The task of the CI server is to control the continuous integration process. The server as a central part of this process is responsible for starting a build of the project at every commit. It informs the developer about the result of the commit using the feedback mechanism.
- The build script automates the build process. The CI server needs a mechanism to run the whole build process of the project after every commit. The build script is responsible for this task. It automates the build process so that no manual interaction of a developer is needed to build the project.
- The feedback mechanism informs the developer about the results of the build. The developer will be informed via email, for example. If the build is broken by a commit of a developer, this developer is responsible for fixing it as soon as possible.

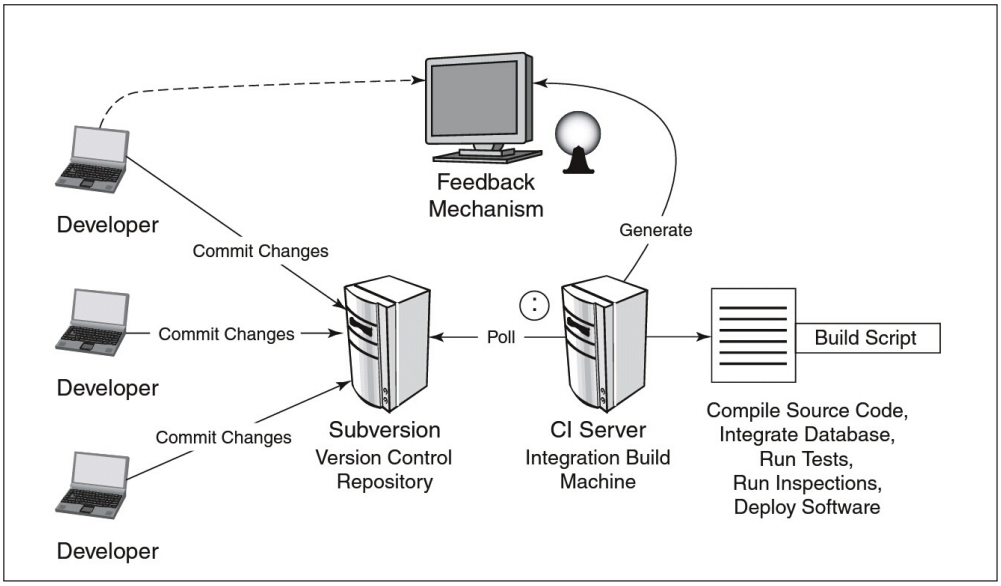


Figure 2.1: Parts of a continuous integration system [8, p. 15]

2.1 Version Control Repository

A version control repository is an archive where among other things the source code, documentation and configuration files of the project are stored. This archive is usually located on a server and every developer of the project has access to this server. It is designed for concurrent read and write access of several clients. Clients are, for example, developers. With a write operation, a client makes his change available to the other clients. With a read operation, the client gets the changes from the other clients.

The special feature of a version control repository is that each of the changes is remembered by the server. Every write operation creates a new so called revision. A revision is the state of the software at a given time. With a version control system, it is possible to receive any of these revisions if required. A revision contains information such as:

- Which files the repository contains at a certain point in time.
- Which developer has changed a file and which lines in the file were affected.
- How many developers have worked on a file and which lines were modified by which developer.

With this information the contribution of the developers from different parts of the project can be covered.

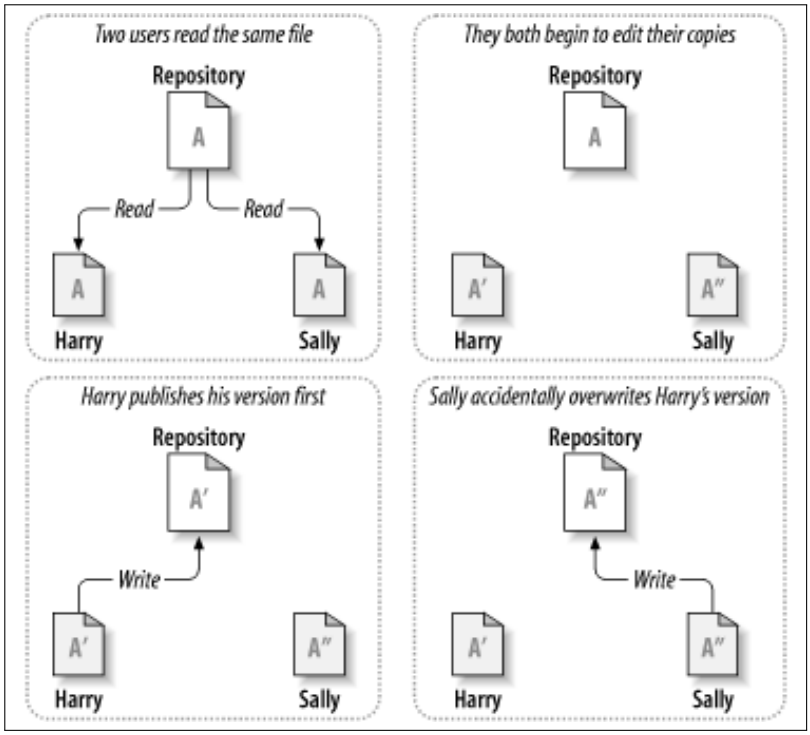


Figure 2.2: Concurrent editing of a file [7]

2.1.1 Concurrent Editing of a File

A version control system supports the developers when they are working together on the same files. One of the problems that can occur when more than one person is working on a file is the possibility of conflicts. A conflict happens when two or more developers are changing the same line of a file in a different way. This scenario is shown in Figure 2.2 [7].

Harry and Sally are both reading the same file from the repository. Both edit this file on their local machine. They each change the file in a different way. Harry is the first to perform the write operation to the repository. He does not have any problems, because at that time the repository is not aware of the other version of the file changed by Sally. The problem occurs if Sally finally writes her changes to the repository and accidentally overwrites Harry's version of the file [7].

There are different solutions for this problem. They are described in detail in the following sections.

2.1.2 The Lock-Modify-Unlock Solution

This solution prevents the clients from modifying the same file at the same time. A file can be read by all clients at the same time, but only one client

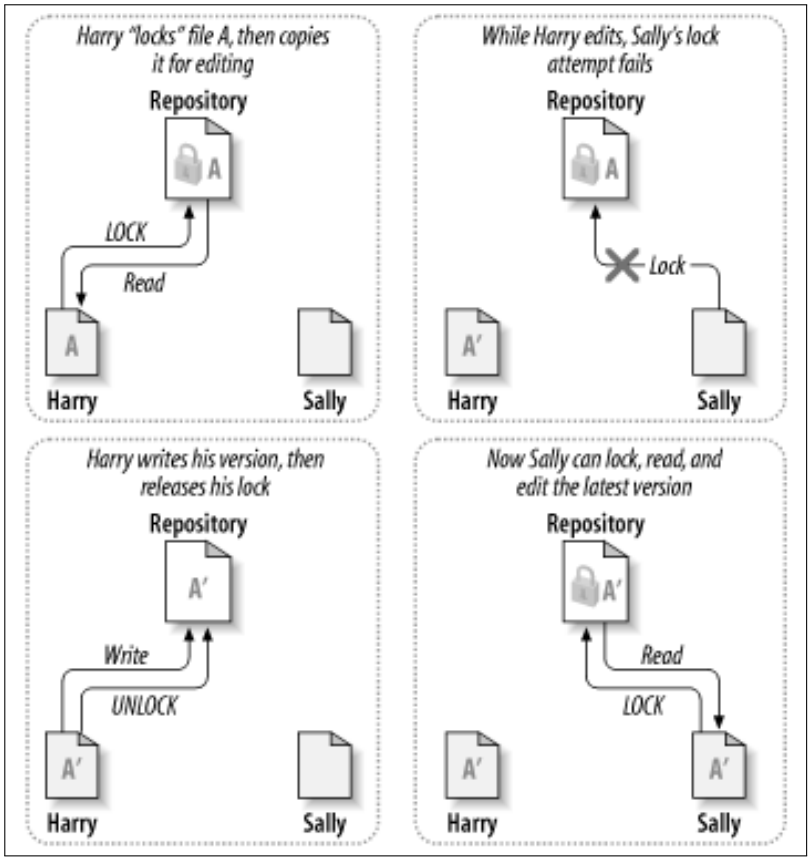


Figure 2.3: Lock-modify-unlock solution [7]

is allowed to write this file.

This is achieved by a lock mechanism on the repository. If a client requests a file for writing a lock on this file is acquired. This lock can only be acquired once. Other clients cannot acquire a lock on the file at the same time. The lock is released, if the client who holds the lock commits his changes to the repository. Then other clients can make changes following the same steps.

As an example Figure 2.3 shows the lock-modify-unlock solution with two clients. Harry reads the file from the repository and acquires a lock on that file. If Sally now tries to lock the same file, it fails because the lock is already acquired by Harry. When Harry performs a write operation to the repository the file is unlocked. Now Sally is able to read the file, acquire the lock and make changes to the file.

There are several limitations to this solution, as described in the following:

- If one client has locked a file, any other client is not able to edit that

file. This means that every other client has to wait until the change is completed and the lock is released. This can quickly lead to problems, especially if the file is locked for a long time. For example, if someone has locked the file and forgets to unlock it then the file may be locked for days with nobody working on it.

- It may happen that a lock is acquired that is not needed. This is the case when two clients edit the same file but different and independent lines of that file. Then a lock is acquired but not needed because no conflict occurs.

- The same may happen the other way around. It is possible that two files are affected by each other, but only one is locked. Then the lock mechanism does fail. A lock is not always an absolutely safe way to prevent conflicts. The clients always have to take care of syntactic correctness themselves. The lock-modify-unlock solution can lead to a false sense of security.

2.1.3 The Copy-Change-Merge Solution

The other solution for handling conflicts is to allow every client to read the file and make a copy of that file in the local working directory. Each client has a copy of the file and is allowed to make changes.

The first client who writes the changed file to the repository server has no problem. The repository server does not know about the changes made by the other clients. When other clients write their changes, they must perform a merge operation with the new version on the server. This merge can only be done by a human, because the semantic correctness must be ensured. The merge operation is the most critical part in this process. If there are only a few changes to the file it is an easy task, but if there are a huge amount of changes from many different clients, the complexity of the merge operation is increased.

To keep merging easy, the merge operation should be performed as often as possible. Then, the differences between the various versions of the files are smaller and conflicts are less likely.

In Figure 2.4 the workflow with the example from above is shown. Harry and Sally are both reading the same file from the repository. They have a copy of that file on their local machine. Both change the file in a different way. Sally publishes her version without any problems. Then Harry tries to perform the write operation but an out-of-date error occurs because the file on the server is not the same as the one he had read before. The latest version of the file on the server contains Sally's changes. Harry must merge the different versions of that file by comparing the latest version from the server to his own. The newly created merged version of the file is published

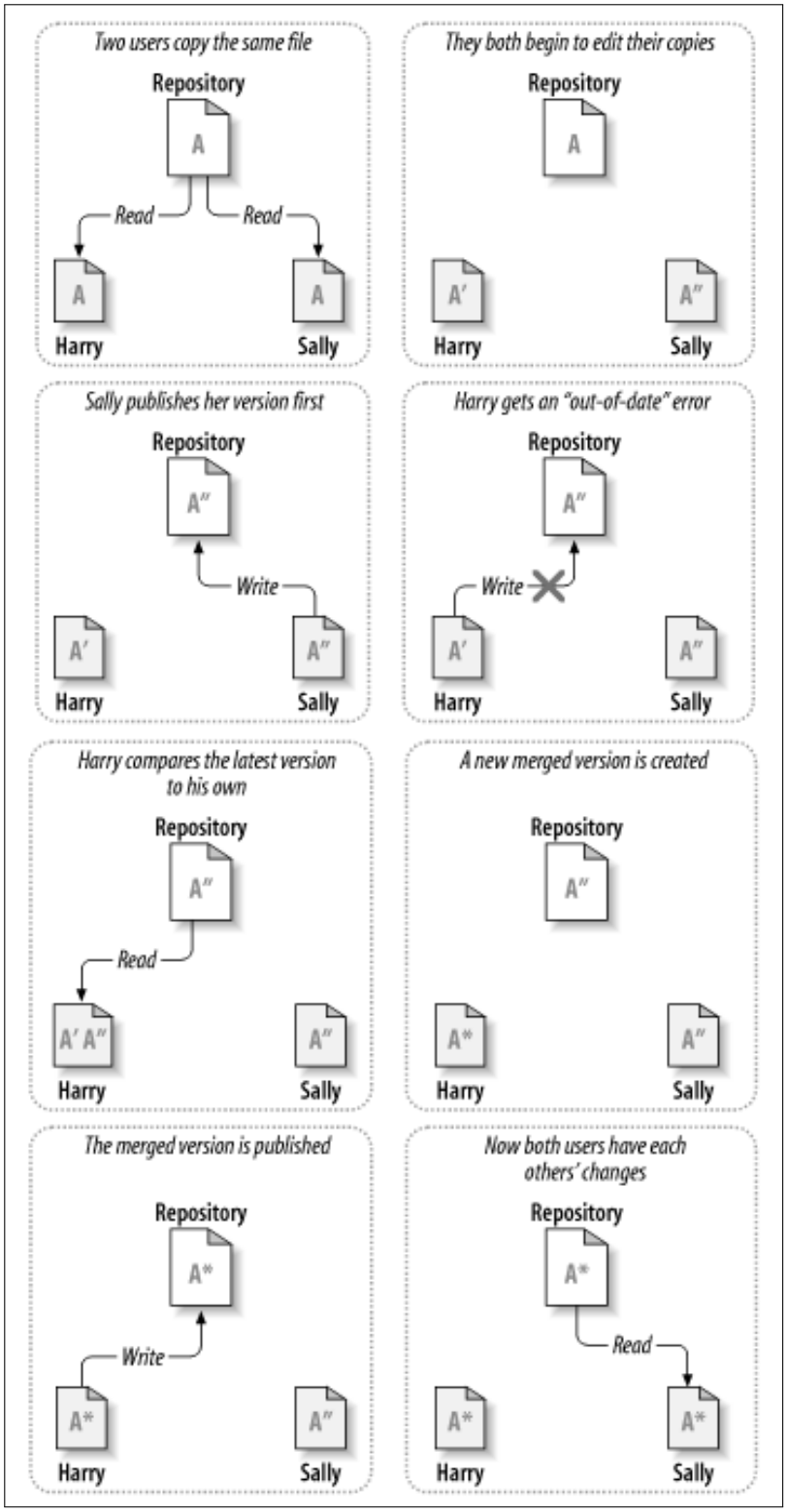


Figure 2.4: Copy-change-merge solution [7, p. 5]

to the server and the next time Sally reads this file, she also receives Harry's merged version [7, p. 4f].

This solution sounds a lot more complex than the lock-modify-unlock solution, but in practice most of the problems generated by other solutions are solved.

2.1.4 Advantages of the Copy-Change-Merge Solution

The advantages over other solutions that allow concurrent editing of files:

- The different clients are able to work in parallel. No one has to wait until other clients have finished the work and the file lock is released. If a client is working on a file for a long time, it is in his interest to read and merge with the new version from the server more often to prevent a large merge operation at the end of his task.
- If two clients edit the same file but at different locations, they can work in parallel and a conflict never occurs. A merge with no conflicts is no guarantee that the program works as intended. To check the semantic correctness of the program, tests are needed.
- The clients are always responsible for merging the changed files by themselves. The false sense of security is reduced.

But nevertheless, there are situations where the locking solution is reasonable. The copy-change-merge solution is based on the assumption that most of the files in the repository are text based files such as source code. If there are binary files such as pictures, this solution does not make sense as the merging of two different versions of a binary file is very difficult or even impossible.

Most version control systems use the copy-change-merge solution but also provide a way of using a locking mechanism for binary files.

In general the communication of the team is important to reduce conflicts. The version control system can assist but not force the communication of the team [7, p. 6].

2.1.5 Centralized Version Control Systems

There are two types of version control systems, which differ mainly in the way they store the data in the repository. The first is the centralized version control system, which stores all informations in a single place 2.5. All information about the revisions are stored only in a central location. The system is designed as client-server architecture, so all communication directed at the repository can be done via the network. The two most popular open

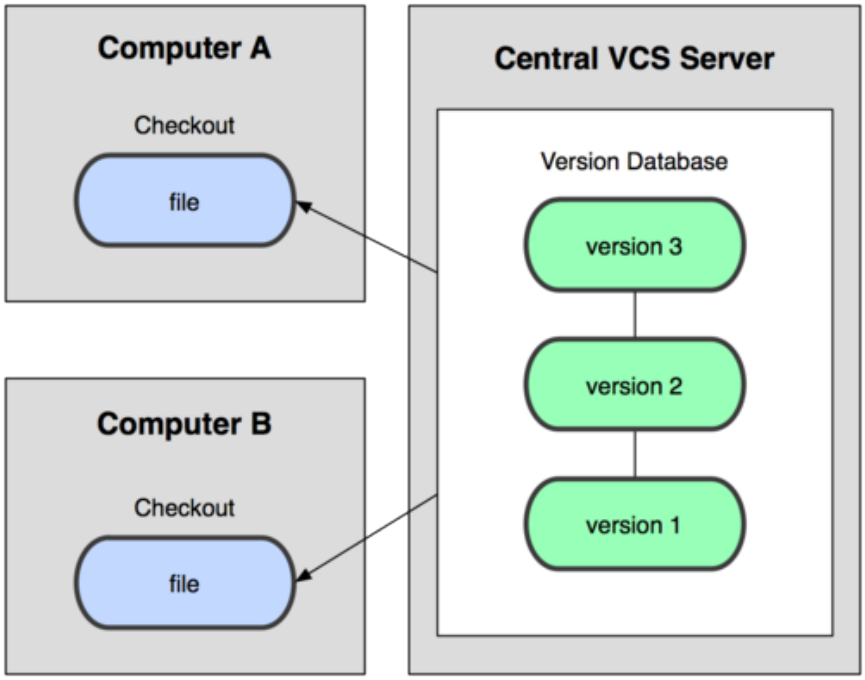


Figure 2.5: Centralized version control diagram [6, p. 2]

source centralized version control systems are the Concurrent Versions System (CVS) and Subversion[1] (SVN). Subversion is a further development of CVS[2] which is no longer under active development.

2.1.6 Distributed Version Control Systems

There have been further developments of the common centralized version control systems. In a distributed version control system, the history data of the revisions is not stored in one single place but in several locations. Every client has its own local repository which contains the data from the remote repository and is used to perform local commit operations. A copy of the repository including all former revisions is stored on each client as shown in Figure 2.6. The client commits to their local repository and pushes the changes to the remote repository later. Popular open source distributed version control systems are Mercurial[3] or Git[4].

Distributed version control systems can also deal with more then one remote repository. Groups of developers can collaborate within the same project.

[1] http://subversion.apache.org/
[2] http://en.wikipedia.org/wiki/Concurrent_Versions_System
[3] http://mercurial.selenic.com/
[4] http://git-scm.com/

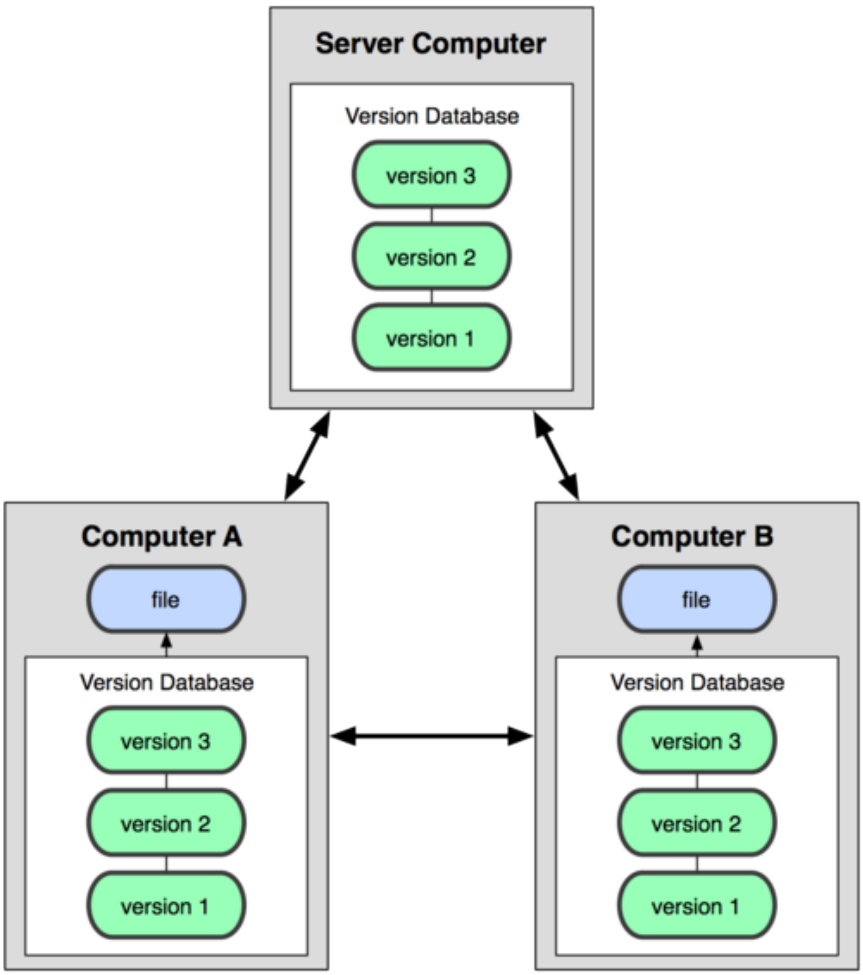


Figure 2.6: Distributed version control diagram [6, p. 3]

Furthermore it is possible to use such systems completely without a remote server. Then only the local repository is used to keep track of the files [6, p. 3].

Distributed version control systems do have a few advantages over centralized version control systems:

- All the history data is available on every client. This means that every client has a complete backup of the repository on his local machine.
- For most of the operations no network is needed. This has the advantage that the operations are faster because no network is involved. Each client can also work with the local repository without an internet connection.

- Using more than one remote repository allows more specialized workflows.

The disadvantages compared to centralized version control systems:

- During the first clone of the repository, more data is transmitted compared to a centralized version control system. This is because all history data of the repository is also transferred to the client. Therefore the first checkout needs more time.
- Most distributed version control systems do not have a locking mechanism which can lead to problems if there are binary files in the repository.

2.2 Continuous Integration Server

The continuous integration server is the central part of a CI system. It is responsible for checking the mainline of the repository for changes and starting a build if changes are detected. Usually the CI server polls the repository every few minutes for changes.

When the build process is finished, the CI server generates reports of the build results and informs the developer by using the feedback mechanism.

The results of previous builds are stored on the CI server. They can be accessed by the developers if needed and are used to generate statistics about the evolution of the project over time, for example, the number of successful tests or the total number of tests over time.

The binaries of successful builds are also stored by the CI server. This makes it possible to always provide the latest working binary of the software.

The CI server provides an interface to configure the build steps. This interface is usually web-based and can be accessed via a common browser.

There are many different CI servers available on the market. Even though all of them offer a similar core functionality, they differ in the additional features they provide. One such server, the Jenkins Continuous Integration Server, is described in the following section in more detail.

2.2.1 The Jenkins Continuous Integration Server

Jenkins[5] is one of the most used open source continuous integration servers. Originally Jenkins was known as Hudson[6]. It was developed by Kohsuke Kawaguchi when he worked at Sun Microsystems[7].

[5] http://jenkins-ci.org/
[6] http://hudson-ci.org/
[7] http://en.wikipedia.org/wiki/Sun_Microsystems

Figure 2.7: Clean Jenkins installation [20, p. 16]

In 2009 Sun Microsystems was purchased by Oracle[8] and in 2011 the project was forked into two separate projects. One was the original Hudson project which was under the control of Oracle and the other was the Jenkins project which remains as an open source project. Jenkins is under active development by many of the former Hudson developers. But even if there are some differences, both projects are in fact very similar [20, p. 3ff].

2.2.2 Jenkins Plugins

Jenkins is written in Java[9] and provides more than 400 plugins which can be used to improve the build process. These plugins make Jenkins very flexible and the area of applications that can be built is not limited to projects written in the Java programming language. With the use of the plugins it is also possible to work with .NET, Ruby, Groovy, Grails, PHP or other projects. The plugin system of Jenkins is powerful and easy to use. Plugins can be installed with one click on the graphical user interface(GUI) of Jenkins [20, p. 3].

2.2.3 Configuring Jenkins

Jenkins provides a web-based GUI. Figure 2.7. shows what the interface of a clean Jenkins installation looks like. Jenkins can be completely controlled via this web-based interface. This includes the complete core configuration

[8] http://www.oracle.com
[9] http://en.wikipedia.org/wiki/Java_(programming_language)

of Jenkins, the plugin management and also the configuration of new Jenkins build jobs [20, p. 3ff].

2.2.4 Jenkins Community

Jenkins has a very large and active Community. They provide mailing lists, IRC channels, development blogs and more. There are weekly releases which contain small updates and bug-fixes. The development pace is very high [20, p. 3].

2.3 Test Driven Development

Test driven development (TDD) is a software development process which is not directly part of CI but it works well together with continuous integration.

Automated testing is very important in a CI process. With daily commits to the mainline, automated tests are imperative to ensure that new code does not break existing functionality. TDD works well with CI because small iterative development steps are used, which is also best practice in a CI process.

2.3.1 Test-First Programming

When working with test driven development, the test is written first and then the production code is composed.

At the beginning of a development task a test is written which checks if the new functionality works as intended. When executing this test it must fail because at that time no production code exists. Then production code is written until all tests are successful [21, p. 17f].

Advantages of test-first programming are described in the following:

- The production code that is generated has a higher quality and it is easier to refactor existing code because there are test cases for every part of the program.
- Writing test first is more challenging as it corresponds to a creative step in the design of the system. It corresponds more closely to a specification step then to a quality control step.
- It minimizes the tendency to write too much code which is not needed for the required functionality. It allows to more easily concentrate on the work that needs to be done at that particular moment without wasting time on writing additional code which is often not needed later.

- Test-first programming guarantees that tests are written at all. Planning to write tests after the production code often leads to situations where it becomes more urgent to write new functionality first and relegating the testing to a later time, which actually will never come.

2.3.2 Test Driven Development Cycle

The test driven development cycle contains the following steps as described in [3, p. 7]:

1. Add a little test: A test is added which checks the functionality of the required feature.
2. Run all tests and fail: At that time no production code exists, so the new tests fail.
3. Make a minor change: A minor change to the production code is made which should make the tests pass.
4. Run the test and succeed: If the change to the production code is correct, all tests are successful. If one or more tests are still failing we go back to point 3.
5. Refactor or remove duplications: To keep the production code at a high quality, refactoring is important. No new functionality is added to the production code. The goal is to keep the production code at a high level of quality in terms of readability and code style. As there are tests for all parts of the code it is not possible to destroy working functionality as long all tests stay successful.

The test driven development cycle uses small iteration steps in order to obtain quick results. This results in many iterations that the developers have made within a day. This makes TDD perfect for use in combination with continuous integration because in CI it is important that developers are able to commit to the mainline often.

Chapter 3

Practices of CI

The following practices were defined by Martin Fowler [9]. They describe what is needed for an efficient continuous integration process.

3.1 Maintain a Single Source Repository

All source code as well as other project related material should be in one single well known place. Version control systems are designed to achieve that. Such a system should be part of every software project. Every developer should know were to find this repository so that he is able to get all files needed. Version control systems are described in detail in Section 2.1. There are good quality open-source version control systems available.

3.1.1 What should be in the repository?

The repository is not only a place to store the source code of the project. Also other project related data can be stored there.

- Of course the source code of the project should be in the repository. Every developer should commit new code frequently. The newest version of the source code is always available for every developer.
- All kinds of scripts which are necessary to build, install and test the software should be added to the repository. These scripts are also needed by the CI server to run the build. If these scripts are in the repository they are automatically version controlled and backed up. So it is possible to get an older revision and build an older version of the software with the proper scripts.
- Adding property files of the software or from the used IDE to the repository helps to share the configurations between the developers.

- Including the third party libraries that are needed ensures a rapid project setup when a developer fetches the data from the repository the first time.

All things needed to perform a complete build should be in the repository. This ensures a fast development setup of the software project.

But there are things that make no sense to have in the repository. This includes things which are large, complicated to install and stable. The operation system or runtime environments are examples that should not be in the repository [9].

3.2 Automate the Build

A build includes everything from compiling and linking the sources, running a test to ensure the software works as expected and running a code analysis to check the code quality. It is important that this whole build process runs automatically. No manual intervention should be needed for the build.

When the build process is done automatically, every step is done the same way every time a build is performed. Also the possibility of failures during the build process is reduced if an automated build script is used.

Every language does have its own automated build environment that can be used to automate the build. In the Java programming language for example Apache Ant can be used to build the software. Ant is designed to be cross-platform, easy to use, extensible, and scalable. It is well suited to automate the whole build process of Java projects. There is also a version of Ant which works with .NET environments. This .NET related version is called NAnt [16, p. 5f].

Often an IDE is used to build the software or a part of the software. In Java for example Eclipse is a widely used IDE which has a lot of build tools included. This is a convenient way to build the software in the development phase.

It is important that we also have a build script that is able to build the software without an IDE. When we build the software on a test server we don't have an IDE installed. In this case a script is needed to do the job.

An automated build process is important to allow the CI server to trigger a build after every commit to the mainline [20, p. 2, p. 5].

3.3 Make Your Build Self-Testing

To check whether the software behaves correctly, we need to test it. Software tests are an integral part of a continuous integration process. They are executed within the build process, and the developers are immediately informed of the result.

There are different testing methods and different testing levels. For the continuous integration process an agile test method is preferred because the test are written first. Test driven development work particularly well with CI. Test driven development is described in detail in Section 2.3.

> "Of course you can't count on tests to find everything. As it's often been said: tests don't prove the absence of bugs. [...] Imperfect tests, run frequently, are much better than perfect tests that are never written at all." [9]

This is exactly what the CI process does. All tests are executed automatically in every build. It is necessary that all test are executed and all tests are successful. The goal is to produce self-testing code. If a test fails, the whole build fails.

In CI it is important that the integration process is done frequently. If this is the case, only small parts of the source code are changed in every commit. This helps to find bugs quickly if the build is broken. The developer only needs to debug the source code which was changed by the last commit.

This is essential to get the full benefit of the continuous integration process. If new source code is not added to the mainline for a long time, the process of integration becomes more time-consuming.

It is not always easy to commit the code often. Developers tend to wait until the code is perfect before committing. There are two techniques which help to ensure that code is committed more frequently.

- Keep the changes small. Try to change only a small number of files. Write a test for the small changes and commit the changes to the mainline as soon as possible.
- Commit to the mainline after each task. If the tasks are small and can be finished within a few hours the code can be added to the mainline as soon as the task is finished.

Commits which are added to the mainline every day also force communication within the development team. To commit the code every day means that conflicts must be merged frequently and developers have to communicate with the team to achieve this task [8, p. 40].

3.4 Every Commit Should Build the Mainline on an Integration Machine

To ensure that the code in the mainline is always in good health and errors are detected as soon as possible a build should be triggered after every commit to the mainline.

A build on the integration machine is necessary to ensure that the build is successful on the reference environment. A successful local build which is performed on the developer machine does not automatically mean that the build is also successful on the integration server. It may happen that the developer forgets to add some files to the repository. Then the build works on the developer machine but fails on the integration machine because some files are missing there.

The reference platform for successful builds is the integration machine. The developers are responsible for ensuring their code and all the tests are working on the integration machine. If the build on the integration machine fails, the developer is forced to fix the build as soon as possible. This rule implies that no developer should commit new code to the mainline minutes before he is leaving his workplace. He must be available until the result of the build on the integration machine is available so that he can immediately fix the failures that may occur [8, p. 65].

3.5 Keep the Build Fast

After every commit a build on the integration server is performed. This results in many builds on the integration server every day. When the development team gets larger a fast build gets even more important because more developers are working concurrently with the integration server.

If the build process takes a long time the continuous integration process does not work properly. The developers need quick feedback about the new code they added. A build process which takes hours to complete leads to a slow development process because the developers have to wait for hours to get feedback from the integration server.

The complete build process should not take longer than ten minutes. If this time is achieved the continuous integration process can work properly [9].

If a build fails it should fail fast. It does not makes sense to run a long running build where the error is detected at the end of the process. Parts which are more likely to fail should be performed at the beginning of the build process. This keeps feedback to the developers fast [8, p. 65].

3.6 Test in a Clone of the Production Environment

To find bugs as early as possible it is important that the environment on the integration machine is similar to the production environment the software later runs on. This reduces the risk of missing errors which only occur in connection with a certain environment.

The test environment should reflect the production environment as much as possible. This includes using the same operating system, installed programs, third party libraries but also special hardware that is used on the production environment.

If the software must run on different environments the tests should not only run on a single environment. Often the same build scripts and test cases can be used which are just executed on multiple integration machines.

Many CI servers support running parts of the build process on different environments. Test runs are a good example to run on different machines. Often they can run in parallel and the results are reported back to the main server. This also speeds up the complete build process when multiple integration machines are involved [9].

3.7 Make it Easy for Anyone to Get the Latest Executable

Developers can easily download the latest binary of the software. That binary is built on the integration server and also tested with automated tests. Every time a developer commits new code to the mainline a new executable is provided. This means that new features or bug fixes are immediately available to all developers and other people related to the project.

If a developer needs a current build of the software for demonstrations, exploratory testing or just to have a look at the new features, he doesn't need to perform the complete build process on the local machine but can easily download the latest executable.

But not only the developers are interested in the latest executable of the software. All project related people should be able to download the latest stable version from a well known place [9].

3.8 Everyone Can See What's Happening

In the continuous integration process it is very important that everyone knows the current state of the software in development. The build should break as infrequently as is possible and errors in the build process should be fixed quickly. In order to achieve this, it is important that a broken build is highlighted to everyone in a very clear way.

An example of this could be a signal light which can be seen by every developer where green indicates a stable mainline build and red indicates that the mainline is currently unstable. When the red signal light is shown, everyone knows that someone has broken the build which means the build should be fixed as soon as possible so that a working version of the software is in the mainline of the repository.

To obtain more information about the errors in a broken build it is useful to have a summery of the last commit including a list of files which have changed. This information can help to fix a broken build quickly.

This kind of information is usually available in a web-based application. This has the advantage that people who are not located in the developer room can have access to data from the latest builds [9].

3.9 Automate Deployment

The build script is responsible for compiling the software and executing the automated tests. In addition to this automated build process the deployment of the software should also be automated.

The benefit of an automated deployment process can be seen clearly when the software is tested in different environments. The binaries to test must be deployed on the different test servers. If that task is done manually there is the risk that errors are made particularly if the task is to be performed often.

If deployment is fully automated it is ensured that the deployment process is performed in the same way every time. It is also possible to start the deployment process automatically after every successful build. This ensures a complete build process which can be performed, fully automated, by the CI server [9].

3.9.1 Continuous Delivery

A further step of CI is the use of continuous delivery. Continuous delivery is about always keeping the software in a releasable state. The whole release process is automated and the software can be released at any time.

As in CI where the integration process becomes a nonevent, the goal of continuous delivery is to make the release process a nonevent. Using continuous delivery it is possible to make new features or bug-fixes immediately available to customers [14, p. 347].

Part II

Practical Part

Chapter 4

Introduction to Catroid

Catroid[1] is an on device visual programming language for the Android[2] platform. Android as described in Chapter 5 is an operation system for mobile devices.

Catroid is designed especially for children and teenagers but can also be used by people of all ages to learn programming without prior programming skills. Catroid programs are created and executed directly on the mobile device, so no PC is needed. Catroid is an open-source project, the source code is available on Github[3].

Users are able to share their created programs with their friends via the community website. The community website contains all the projects created by the community. All users are free to download, use and remix the existing projects [13] [12].

4.1 Visual Programming Language

Catroid programs are created using a visual programming language. Visual programing can be described as follows:

> "Visual programming predominantly consists in moving graphical elements instead of typing text." [19, p. 1]

As Catroid programs are mostly created by young people the use of a visual programming language comes with several advantages. Visual programming is not easier by itself but is more attractive to young people. They are more motivated to try out this kind of programming style which makes intensive use of graphical elements [19].

[1] http://catroid.org
[2] http://android.com
[3] https://github.com/Catrobat/Catroid/

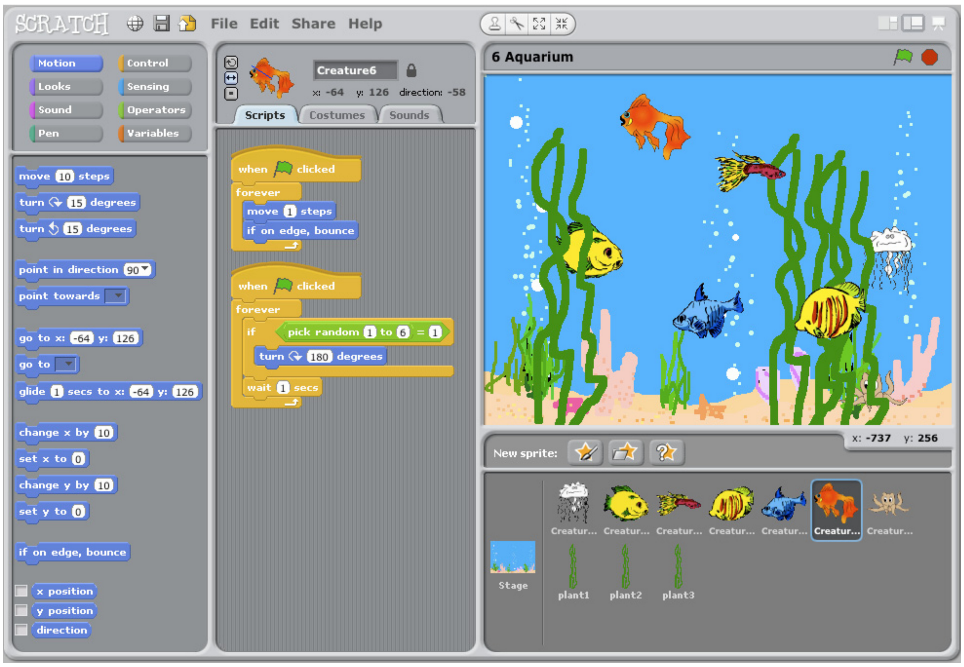


Figure 4.1: Scratch [18]

4.2 Scratch

Catroid is inspired by Scratch[4], a visual programming language for desktop platforms which is designed for children. Scratch was developed by the Lifelong Kindergarten Group at the MIT Media Lab. The aim of Catroid is to make a Scratch like functionality available on mobile platforms. Mobile platforms provide some benefits such as the sensors of the devices which can be used to increase the potential of the created programs [17].

As shown in Figure 4.1 Scratch is using Bricks to build the programs. The Scratch interface is designed to be as simple as possible. With the single-window user interface, all important components are always visible.

Scratch is designed for people with no programming experience. The goal of this program is to learn programming in an easy and playful way. To make programming more exciting, it is possible to work with various kinds of media such as images and sounds in an easy way [18].

The Scratch community is very active and consists of 1,198,546 registered members which have uploaded 2,755,098 Scratch projects to the community website as of 23.9.2012. These numbers show the importance of a community website where the users can share their created programs with their friends along with the rest of the world [15].

[4]http://scratch.mit.edu/

Figure 4.2: Catroid in Action: The left image shows Catroid while creating a program. The right image shows Catroid while executing this program.

4.3 Catroid in Action

The left image of Figure 4.2 shows Catroid whilst building a program on a android device. Bricks represents the statements of the program. The bricks are color coded by category which makes it easier to find the proper brick. In this image the "Turn left" brick is inserted in the program and therefore the brick is highlighted.

The shown part of the Catroid program is executed when the fish sprite is touched. The sprite is rotated and the size is increased. All this actions are described with the use of the visual programming language. When starting the program, all the statements are executed in a sequence. The right image of Figure 4.2 shows the program in the state of execution.

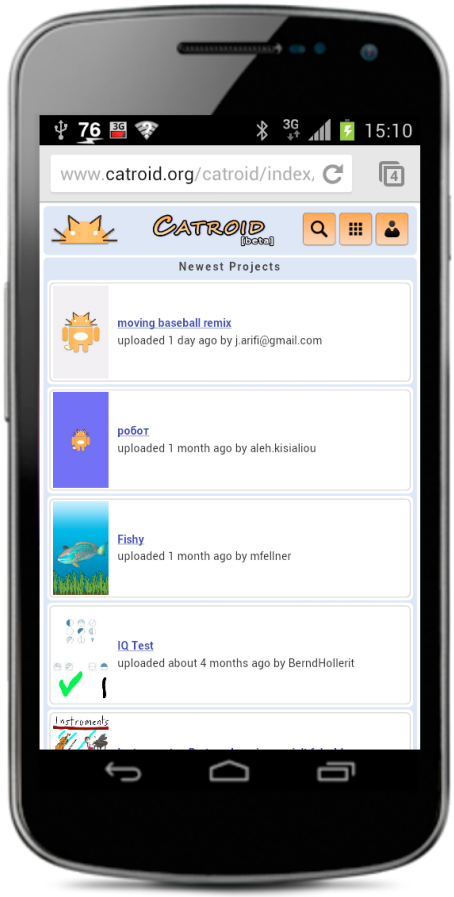


Figure 4.3: Catroid Community Website

4.4 The Catroid Ecosystem

The language of the Catroid program is called Catrobat. Catrobat code is stored in a defined XML[5] document which makes it possible to share and remix projects over different platforms. The Catrobat umbrella project consists of various parts described below.

4.4.1 Apps on Mobile Devices

There are Apps for different mobile platforms in active development. Currently the most advanced mobile App is Catroid which is developed for the Android platform. There are also Apps for other mobile platforms in devel-

[5]http://en.wikipedia.org/wiki/Extensible_Markup_Language

Figure 4.4: Catroid while executing a program to control Lego Mindstorms robots [19]

opment including iOS[6] (developed by Apple[7]) and WindowsPhone[8] (developed by Microsoft[9]). These Apps are in an early state of development, the Android App is leading in terms of usability and functionality.

4.4.2 Catrobat Community Website

With the Catrobat community website users can share, download and remix existing Catrobat projects. Projects which are created on one mobile platform can also be edited and executed on other platforms. For example a catrobat program which is created and uploaded from the Android platform can be downloaded and executed on the other platforms as well.

[6] http://www.apple.com/ios/
[7] http://www.apple.com
[8] http://www.microsoft.com/windowsphone/
[9] http://www.microsoft.com

4.4.3 Control Other Devices Using Catroid

There are many sub-projects in development which expand the functionality of the core Catroid App.

An interesting example is the Lego Mindstorms[10] project which makes it possible to control Lego Mindstorms robots using the Bluetooth[11] technology of the mobile device. Figure 4.4 shows Catroid whilst executing a program which allows control of a Lego Mindstorms robot. When pressing one of the buttons, the proper Bluetooth command is sent to the Lego robot.

There is also a sub-project which allows control of a Parrot AR drone[12] via WLAN[13]. The Parrot AR drone is a Quadrocopter with various built in sensors and cameras.

These and other extensions enables Catroid to act as a remote control device for various external Hardware. These extensions are an interesting feature which makes creating Catroid programs even more fascinating.

[10] http://mindstorms.lego.com/
[11] http://en.wikipedia.org/wiki/Bluetooth
[12] http://ardrone.parrot.com/
[13] http://en.wikipedia.org/wiki/Wireless_Local_Area_Network

Chapter 5

The Android Platform

Catroid is developed for the Android[1] operating system. Android is an operating system and software platform for mobile devices. It is developed by Google[2] and the Open Handset Alliance[3].

5.1 The Android Environment

As shown in Figure 5.1 the Android system is based on a linux kernel. The applications for the Android platform can be written completely with the Java programming language [1, p. 4].

This is possible by using a virtual machine to execute the applications. The virtual machine used by the Android system is called Dalvik and is especially designed for mobile devices. When building an Android application the Java bytecode is converted to the Dalvik format and then executed in the Dalvik virtual machine which is running on Android devices. Using a virtual machine to execute the Apps makes it easier to run the Apps on various devices with different hardware [4, p. 17f].

5.2 Distribution of Android Versions

Android is currently the most used operating system for mobile devices. It is running on hundreds of millions of devices all over the world. There are more than 100 different devices from many different manufacturers available on the market [11].

The Android operating system is constantly evolving and new versions are frequently published. The update speed of software installed on the devices on the other hand is very slow. Therefore there are lots of devices which are running old versions of Android. Currently there are 11 different

[1] http://www.android.com/
[2] https://www.google.com/about/company/
[3] http://www.openhandsetalliance.com/

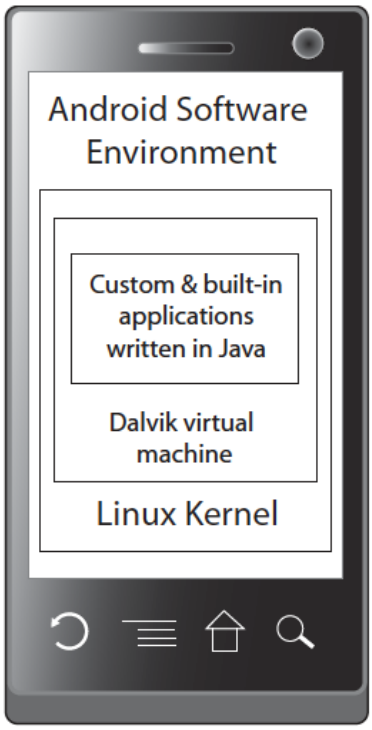


Figure 5.1: The Android Software Environment [6, p. 3]

Android versions which are actively used by customers. That means that Android applications that are created must support at least the most used versions of Android in order to reach a large proportion of the customers [10].

The downside of that huge number of different devices with different versions of Android is described below.

5.3 The Fragmentation of the Android Platform

The fragmentation of the Android ecosystem comes from the fact that the manufacturers rarely update the operating system of their devices. Their goal is to sell phones, so updating old devices is not their first intent. In addition to the different versions there are also different form factors of the devices available. The devices differ in terms of the display size, display resolution and aspect ratio.

Furthermore, the manufacturers of the devices such as Motorola[4], Sam-

[4]http://www.motorola.com

sung[5] or HTC[6] modify the user interface to differentiate themselves from each other. The goal of the manufacturers is to create a unique user experience on their devices to stand out from their competitors.

This fragmentation of the Android ecosystem makes it more difficult to test the applications in an automated way. User interface testing is particularly hard if the tests should run on every device with all the different versions of Android.

[5] http://www.samsung.com
[6] http://www.htc.com

Chapter 6

Introducing CI in Catroid

When Catroid development began, continuous integration was not used. Every developer wrote the task on his local machine and merged the changes to the mainline if the tests were successful on his environment.

6.1 Catroid Without CI

As the Catroid project rapidly grew and more and more developers were involved in the development process, more people were actively working on the same codebase. This presented some problems:

6.1.1 Manual Processes

Many steps have to be done by hand, if a task is finished and the developer wants to merge the change in the mainline. The problem of these manual steps was that they were not always executed as intended. Every developer compiled the project with his own IDE, and test runs were also started manually.

As more developers were added to the project these manual tasks resulted in more and more problems. Once a developer had finished a task, he executed the tests manually and if he thought everything was ok, the change was merged to the mainline of the repository. All this was done manually, so if the developer had forgotten to execute some tests, a possible error was merged to the mainline and would stay there for a long time.

6.1.2 Late Error Detection

Errors were often detected too late. For the same reason as above, errors were often detected long after they were caused. In the meantime other developers had already used the faulty code from the repository and fixing these errors was more difficult and time consuming.

6.1.3 Fragmentation of the Android Ecosystem

There were more and more different Android devices available. This fragmentation of the Android ecosystem 5.3, resulted in unstable tests. The user interface tests became particularly unstable because the user interface differed slightly on each of the various devices.

This was a problem as the significance of the test was lost. Some tests were successful on a device from one developer but were failing on other devices. Getting the test working on every device was very difficult, as a huge number of different devices with different software versions were on the market.

A mechanism was needed to ensure that all tests were successful. With all the different devices it was difficult to ensure that the software was working as intended using automated tests.

6.1.4 Android Emulator

The Android Software Development Kit (Android SDK) provides an emulator on which the software including the tests can be executed without a physical device. This emulator can be configured as needed in terms of display size, pixel density and Android version. But nevertheless, this emulator has a few disadvantages:

- The emulator is very slow compared to a real device. The execution of the tests takes much more time.
- The emulator does not support the OpenGL version which is needed to execute all tests of Catroid.
- It is not possible to simulate all communication options of a real device. For example Bluetooth is not supported by the emulator. For tests which need such hardware components, a physical device is needed.

These facts make it hard to use the emulator for testing. There is always a need for physical devices to execute all tests.

6.2 Introduction of Continuous Integration

The introduction of CI in Catroid addresses the problems mentioned above. The main features of CI do solve most of the problems and in addition they also help to solve problems which are not directly related to CI.

The problem with the fragmentation of the Android ecosystem with the many different devices from different manufactures was easier to handle as the integration server acts as a reference platform. On our CI server a physical device is directly connected and acts as the reference platform where all the tests must be successful.

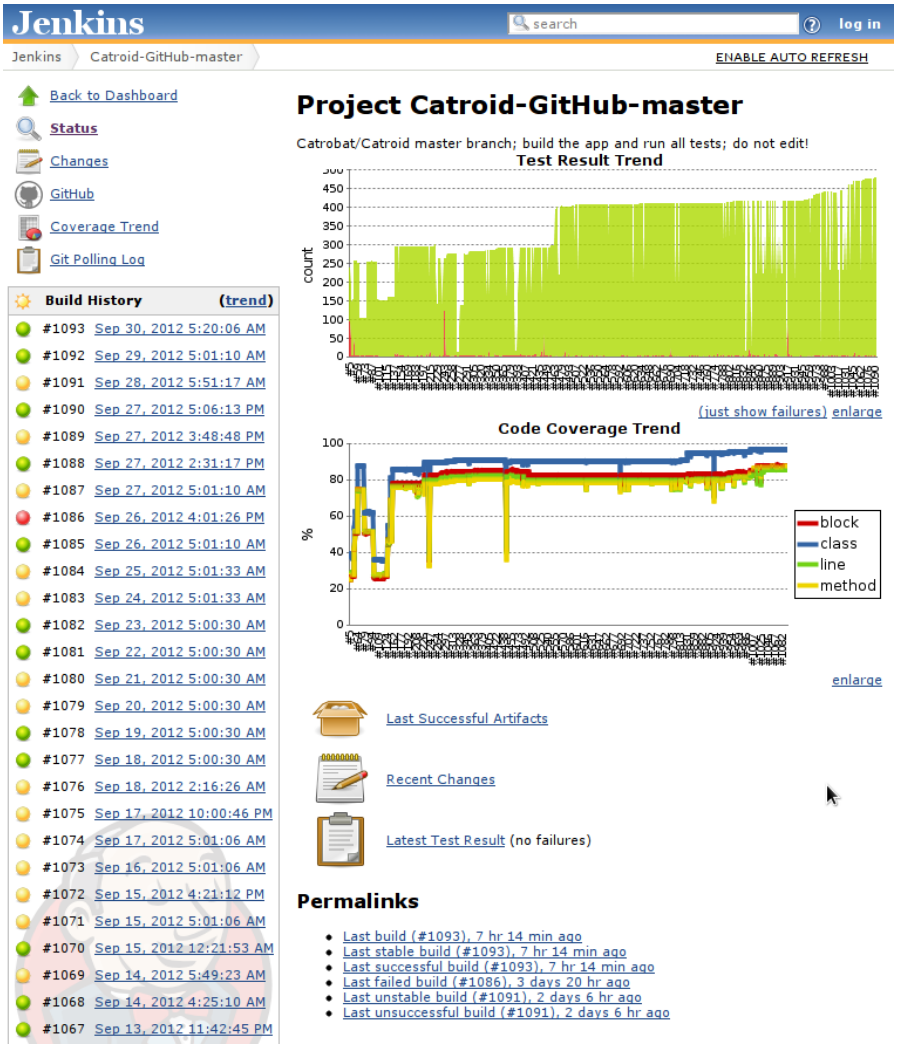


Figure 6.1: The CI server Jenkins shows an overview of Catroid

With a physical device it is also possible to use all communication options such as Bluetooth. Tests which use Bluetooth are executed in a defined environment on the integration server and it is possible to run more complex test cases which require different components to interact with each other, for example a Bluetooth test executed on the device that interacts with a Bluetooth server running on the integration server.

With the build scripts the CI server is able to perform the whole build process automatically. For this project the CI server Jenkins is used.

6.2.1 CI Server Jenkins in Catroid

The CI server builds the executable binary, the tests are executed and some code measurement tools are involved. With this level of automation it is possible to perform a complete build after every commit to the mainline of our project repository. After a successful build the CI server provides the latest binary for download. Everyone can install this binary in order to always have the latest version.

Figure 6.1 shows the web interface of the CI server Jenkins used in Catroid. On the overview page important data of the project are shown:

- On the left the build history of the latest builds is shown. The result of the builds is color coded to get a quick overview:
  - green means a stable build with no failures
  - yellow means an unstable build with one or more failing unit tests
  - red means that an error in the build process occured
- On the right top the trend of the test results is shown. On the ordinate axis the number of executed tests are shown. They are also color coded with the same colors as above. Currently 479 test are executed during the build process.
- The image below shows the trend of the code coverage. The proportion of code that is tested is growing over time.
- The artifacts of the last successful build contains the actual binary of Catroid.

6.3 The Integration Process in Catroid

To ensure the high quality of the source code an integration process is defined. Every developer who is adding new functionality or bug-fixes to the project must follow this process. The goal is to make sure that all source code which is merged to the mainline is readable, tested and of high quality. The code is reviewed by a second developer who was not included in writing that code. In this review the second developer checks if the code passes all the rules that have been defined.

In the following the integration process as shown in Figure 6.2 is described:

1. First a feature branch is created to implement the task. Using a branch comes with the advantage that changes can be pushed to the remote repository without merging the changes to the mainline.

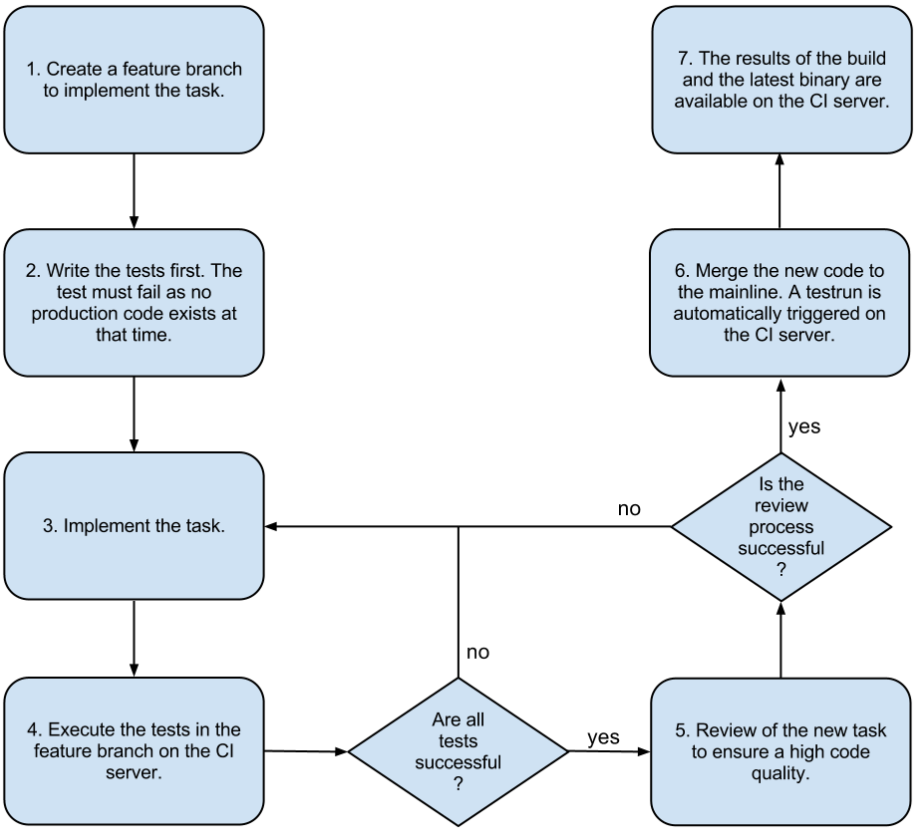


Figure 6.2: The CI process used in Catroid

2. Tests are written first to check the correctness of the implementation. The new tests are executed and will fail because no production code exists at that time.

3. Once the tests are written the developer starts writing the production code. It is also possible that more than one developer is working on a task but usually the tasks are small enough for one developer to handle it. Large tasks are split into several smaller tasks if possible.

4. The developer is not allowed to commit the code to the mainline if the integration process is not finished. With that branch the developer can commit his changes to the repository without changing the mainline of the repository. If the implementation of the task is finished, the developer runs all tests on the test server to check if the software works as intended on the reference platform. This test run is performed on the feature branch of the repository in which the new task is developed. At that time the code is not in the mainline.

5. If all tests are successful, the next step is the review process. A different developer who is not involved in the development of that task reviews the code. He checks if the written code is readable, tested properly and if all tests run on the test server. When failures are detected in the review process, the developer has to correct them before he can go further.

6. Once the review process is finished properly, the next step is to merge the new code into the mainline of the repository. When the CI server detects that new code is in the mainline, a test run is automatically started.

7. On success, the tested binary is available on the CI server. As the CI server is open to the public, everyone is able to download the latest stable version of Catroid. This may happen more than once a day which means that new versions of Catroid are available frequently.

With this process only tested code which has been seen by at least two people is added to the mainline. Each task is self-contained, so every time a task is finished and merged to the mainline, the build on the CI server is a new version with a new feature or a bug fix.

Chapter 7

Measuring CI Capability of Catroid using the CI Grid

A way of measuring and improving the capability of continuous integration is using the CI grid. The CI grid gives a quick overview of the CI capability of a project.

7.1 The CI Grid

With the CI grid it is possible to visualize the CI capability of a project or compare the capability of different projects in an organization. This allows the identification of problems in the CI process or the ability to share practices between the different projects of an organization. The Grid does not measure if the developers are doing CI properly, it measures if the environment and the capability to use CI is available [22].

7.1.1 Representation of the CI Grid

For the visual representation of the CI grid a matrix is used. The data to fill the matrix is generated by asking questions concerning the various parts of the CI process and the answers are color-coded in the matrix. Using colors not numbers to fill the matrix comes with the benefit that a quick overview is possible. To get an overview the data does not need to be very precise [22].

7.2 Questions for the CI Grid

To fill the grid, questions are asked and the answers are color-coded in the matrix. The meaning of the four different colors are: [22]

- Red: Not started.

- Yellow: We are working on it.
- Green: Yes, we have it.
- Blue: We have no interest in this.

The questions are grouped into three categories which are described in the following:

7.2.1 Daily Build Questions

The questions in this category deal with the tasks in the daily build process. The following questions are taken from [22]:

- Compilation: Is the software compiled every day automatically? Compiling and linking of software should be fully automated. The whole software product with all subcomponents must be included in the compilation process.
- Unit testing:
  - Are the Unit tests executed automatically after the software is compiled? No manual steps should be needed to execute the Unit test.
  - Is it easy for developers to add a new Unit test? Adding unit tests by putting them into the version control system is a common practice.
- Monkey testing: Are monkey test executed as part of the build process? Monkey tests are automated tests which use the application with random input. No specific tests are defined, the input of the monkey tests is generated randomly. Monkey tests check if the application can handle all kinds of inputs without crashing or freezing the system.
- Installation: Is the software installed to a production like environment within the daily build process? This does not mean that the software needs to be deployed to the public every day. It should be installed in an environment which is similar to the production environment. The installation to the production like environment should happen automatically.
- Reporting:
  - If a developer breaks the build, is he automatically informed of the error? This does not mean that an email should be sent to every developer if a build is broken. If an email is sent to every developer, they tend to ignore them because they think its not their fault.

  - Are reports about the build process automatically generated and available to the developers? Report generation should happen automatically and access to the reports should be possible for all developers.

- Policy: Does repairing a broken build become the first priority for the developers? It is important to always have a working mainline of the project repository. If failing builds become normal then the whole daily build process become useless as the significance gets lost.

7.2.2 CI Questions

The CI questions verify which tasks are included in a CI process. The following questions are taken from [22]:

- Anytime integration: Can any developer add new code to the mainline of the project without too much effort? Frequent commits to the mainline are important for a CI process to work well. To achieve that, the process of adding new code should be as easy as possible. The mainline of the project means the total product source, not a feature branch in the repository.

- Compilation: Is the software automatically compiled within an hour of a commit to the mainline? If new code is added to the mainline the compilation of the software should be triggered. This can be achieved by polling the version control system for changes in the mainline.

- Unit testing:

  - Are the Unit tests executed automatically as part of the CI build process? After the compilation of the software the execution of the Unit test must be triggered to ensure that added code does not break existing functionality.
  - Is it easy for developers to add new Unit tests? New Unit tests should be executed when they are added to the version control system. No manual configuration on the CI server should be necessary to add unit tests.

- Monkey testing: This is the same as in the daily build section.

- Installation: Is the software installed to a production like environment within the CI process? The installation to a production like environment should be included in the continuous integration process.

- Reporting: These questions are the same as in the daily build section. The result of the last CI build can be visualized using a signal light

which every developer can see. Then everyone is continuously informed about the build state.

- Policy: This is also the same as in the daily build section.

7.2.3 TDD Questions

Test driven development is not directly part of continuous integration but it is a software development process which works well with CI. TDD also uses development in small steps which is important in CI too. The TDD section in the grid shows if TDD is used or not.

7.2.4 Metrics

With the basic metrics it is possible to get an overview of the health of the CI process.

- Integration feedback time: This is the time from the commit to the mainline until the build has finished and the reports are available. If anytime integration is not used this time is the same as the build feedback time.
- Build feedback time: This is the time the whole build process takes on the integration server. If only daily builds are used the build feedback time is 24 hours.
- Test coverage: This is the source line based test coverage. It describes how many lines of code are executed in the tests. If every line of the source code is executed in the tests the code coverage is 100%. This number is a quality measurement of the tests in the project. If the test coverage is very low the software has not been tested thoroughly enough.

7.3 CI Grid of Catroid and Sub-Projects

Figure 7.1 shows the CI grid of Catroid, Paintroid and Catroweb. Paintroid is an image manipulation application especially designed for use with Catroid. Catroweb is the project which implements the community website.

The CI Grid is perfect for getting an overview of which project is using most of the CI practices and to compare the projects with each other.

7.3.1 Analyzing the CI Grid

The Catroid and the Paintroid project are nearly identically represented in the Grid. The only difference are the missing monkey tests in the Paintroid project.

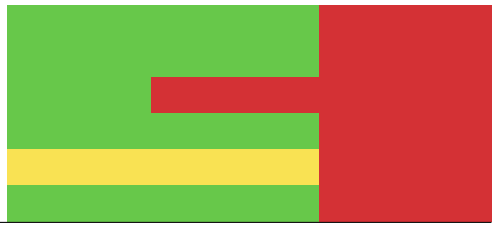
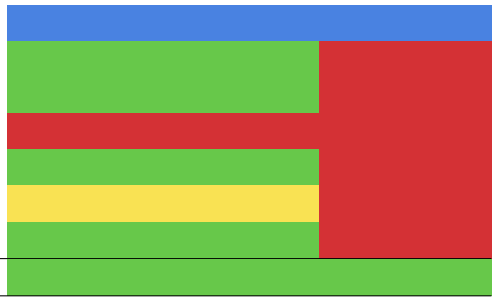
Project	**Catroid**	**Paintroid**	**Catroweb**
Technology (Program Language)	Java	Java	PHP, SQL
Platform	Android	Android	Web
Daily Build			
Compilation			
Unit testing			
Monkey tests			
Installation			
Reporting			
Policy			
Continuous Integration			
Anytime integration			
Compilation			
Unit testing			
Monkey tests			
Installation			
Reporting			
Policy			
Test-Driven Development			
Metrics (Estimations)			
Integration feedback time	75m	17m	?
Build feedback time	75m	17m	?
Test coverage	86%	77%	?

Figure 7.1: CI Grid of Catroid and Related Projects

The reason that these two projects are similar is that both are using the same programming language and the same platform. Introducing the CI process was very similar in both projects. Most of the configuration on the CI server is identical and could be easily transferred between projects.

The fact that the projects appear so similar in the CI grid shows that communication between the teams is working well.

The Catroweb project is currently not using a daily build process or a continuous integration process which is shown as red in those parts of the Grid. The introduction of continuous integration in the Catroweb project is recommended to take advantage of the CI features in that project.

There are some problems of Catroid and Paintroid that are highlighted in the Grid.

Reporting

The reporting process for when a build fails is not adequately resolved. The results are published on the website of the CI server but the developer which has broken the build is not informed individually.

If a developer commits new code in the mainline he must wait until the build has finished on the CI server and then check the CI website to get the result of the build. The preferred way of reporting is that the developer which has broken the build is also informed individually about the error.

Sending an email to every developer is also not ideal. There is the risk that the developers start to ignore these emails as they think they are not responsible for the error.

Anytime Integration

The second issue is the lack of "anytime integration" in the projects. The CI process which is used includes feature branches and a manual review process. This means that a great effort is required to commit code in the mainline of the repository.

As we like to ensure that code which is added to the mainline is seen by at least two people the "anytime integration" is hard to achieve. The integration process currently used works well and at the moment there is no need for anytime integration in the project.

Monkey Testing

Currently monkey tests are only used in Catroid. They should also be introduced in the other projects.

As the monkey test is long running it is only executed in the daily build process. The monkey test runs the application with random input and checks if the application crashes or freezes the system. The result is also published on the website of the CI server.

Chapter 8

The Future of Continuous Integration in Catroid

The introduction of continuous integration comes with many positive impacts for the project. A defined process for adding code to the mainline, a reference platform to make it easier to run UI tests and an easy way for every developer to get the latest executable.

For the future of continuous integration in Catroid and the other projects there are some important elements that must be resolved in a more suitable way.

The suggested improvements are described in the following:

8.1 Reduce Runtime of the Tests

It currently takes a long time to run all the tests. As we are using test driven development the number of tests which are executed is rapidly increasing.

Improving the runtime of the tests is important because the developers need quick feedback about new code they have added to the mainline. If it takes too long, the developers are not willing to integrate into the mainline often because they are loosing too much time waiting for the results of the build.

8.1.1 Catroid

The UI tests for Catroid are particularly long running. Currently more than an hour is needed to execute all the tests. This time must be reduced in order to use CI in a better way. As described in Section 3.5 the recommended time for the tests to run is around ten minutes or less.

Reaching that time can be achieved by running the tests in parallel on multiple devices or emulators. The CI server Jenkins supports parallel running tests as well as tests that run on different machines by using a

master-slave approach. All Catroid tests are independent of each other, so parallel running tests could be easily achieved.

8.1.2 Paintroid

The time for running all the tests of the Paintroid project is currently 17 minutes which is reasonable for the moment. But the number of tests is growing rapidly so a way to run the tests in parallel is also needed in the future.

8.2 Automation of the Integration Process

While the build process of Catroid is fully automated using a build script on the CI server, the process of adding code to the mainline of the repository should be more automated.

Currently there are several manual steps which must be done before new code is added to the mainline of the repository. Manual steps such as switching between branches or manual code reviews take a lot of time and in every manual process there is the possibility that failures could happen.

The CI process currently used ensures that only high quality code which is reviewed by independent team members is added to the mainline of the repository. This comes with the downside that the process of adding new code to the mainline is time intensive and also another team member is needed for the review process.

Considering that every developer should commit his changes to the mainline at least once a day, the process of adding code takes too much time.

8.2.1 Using Pull Requests for a Better Automation of Merges to the Mainline

Catroid is currently hosted on Github[1] which allows the use of pull requests to add new code to the repository.

With a pull request a developer can inform other developers that he has pushed new code to the repository. This code can be reviewed by the other developers remotely. If the pull request is accepted it is automatically merged to the mainline of the repository.

Using pull requests can help to automate the merging process. A manual review is still needed but the review can happen remotely which also has benefits when working with external developers.

[1] https://github.com/about

8.3 Commit to the Mainline More Often

Developers should commit their changes to the mainline more often. This is related to the lack of automation of the integration process described in Section 8.2.

8.3.1 Existing Situation

Currently a developer only commits his code to the mainline every few days or even weeks. New code is only added to the mainline if it is finished and of a high quality. Often it is not possible to achieve that in a few hours or within a day. In addition, the review process must be completed before merging code to the mainline. This leads to situations where finished tasks cannot be merged to the mainline because they are waiting for the review process or related issues. The intensive use of Branches in the development process also leads to less frequent commits to the mainline.

Implementing a task and going through the whole integration process within a single day is difficult. In order to achieve the full potential of the continuous integration process, this situation should be improved.

8.3.2 Desired Situation

The time taken to add new code to the mainline of the repository should be reduced as much as possible. This can be achieved by using smaller development tasks or by pushing changes in the mainline even if the task is not completely finished.

If the changes are pushed to the mainline more often, this integration of new code is much easier because there are only a few changes which need to be merged with the mainline. The goal is at least daily commits from every active developer in the team.

8.4 Add CI to More Catroid Related Projects

Currently continuous integration is not used by all Catroid related projects. To take advantage of CI in these projects it should be also introduced to them.

The CI server Jenkins which is used for Catroid and Paintroid comes with lots of plugins which can also be used to build non Java projects. These plugins include support for IOS, Windows phone and PHP projects which makes it possible to do CI in these projects using our continuous integration server Jenkins too.

8.5 Add Mutation Testing to Catroid and other Projects

Mutation testing is a mechanism which allows evaluation of the quality of the tests used. This is done by adding errors to the program and checking if the tests detect the introduced error.

When adding errors to the program many different versions of that program are created. These versions are called mutants. If the failure of a mutant is detected by a test, this mutant is killed. All mutants which are not killed by the existing tests indicate missing or incorrect tests. The quality of the test can be measured by using this method. Also, places in the code where tests are missing or incorrect can be identified [2].

Currently mutation testing is not used in Catroid or one of the related projects. To take advantage of mutation testing it should be added to the build process. As mutation testing needs much resources, it should not be performed after each commit, but only if resources are available, such as in a nightly build.

List of Figures

Bibliography

[1] W. F. Ableson, R. Sen, and C. King. *Android in Action*. Manning Publications Co., revised edition of unlocking android edition, 2011.

[2] R. T. Alexander, J. M. Bieman, S. Ghosh, and B. Ji. Mutation of java objects. *IEEE Software Reliability Engineering*, 2002.

[3] K. Beck. *Test-Driven Development by Example*. Addison Wesley, 2002.

[4] A. Becker and M. Pant. *Android Grundlagen und Programmierung*. dpunkt.verlag GmbH, 2009.

[5] E. M. Burke and B. M. Coyner. *Java^{TM} Extreme Programming Cookbook*. O'Reilly, 2003.

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

[7] B. Collins-Sussman, B. W. Fitzpatrick, and C. M. Pilato. *Versionskontrolle mit Subversion*. O'Reilly Verlag, 2006.

[8] P. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Addison-Wesley Professional, first edition, 2007.

[9] M. Fowler. Continuous integration. *Martin Fowler*, 2006.

[10] Google. Android dashboards, September 2012. from http://developer.android.com/about/dashboards/index.html.

[11] Google. Android, the world's most popular mobile platform, September 2012. from http://developer.android.com/about/index.html.

[12] T. Gritschacher. A community website for interactive mobile content created by children and teenagers. Master's thesis, TU Graz, 2011.

[13] T. Gritschacher and W. Slany. Standing on the shoulders of their peers: Success factors for massive cooperation among children creating open source animations and games on their smartphones. *Proceedings Int. Conf. on Interaction Design and Children*, 2012.

[14] J. Humble and D. Farley. *Continuous Delivery*. Addison Wesley, 2011.

[15] Lifelong Kindergarten Group at the MIT Media Lab. *Scratch Stats*. http://stats.scratch.mit.edu/community/, September 2012.

[16] S. Loughran and E. Hatcher. *Ant In Action*. Manning Publications Co., 2007.

[17] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick. Scratch: a sneak preview [education]. In *Creating, Connecting and Collaborating through Computing, 2004. Proceedings. Second International Conference on*, 2004.

[18] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The Scratch Programming Language and Environment. *ACM Trans. Comput. Educ. 10*, 2010.

[19] W. Slany. A mobile visual programming system for Android smartphones and tablets. *Proc. IEEE Symposium on Visual Languages and Human-Centric Computation*, 2012.

[20] J. F. Smart. *Jenkins The Definitive Guide*. Wakaleo Consulting, 2011.

[21] P. D. A. Spillner. *Systematisches Testen von Software*, volume 1. dpunkt.verlag GmbH, 2008.

[22] B. Vodde. Measuring continuous integration capability. *CrossTalk*, 2008.