David Reisenberger

Continuous Integration in an Android Open Source Project

Master's Thesis

Graz University of Technology

Institute for Softwaretechnology
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Wolfgang Slany

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

Graz, December 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

Catrobat is a visual programming language that allows users to create and run programs on their smartphone or tablet. The Catrobat team is composed of a large number of students from Graz University of Technology as well as contributors from all over the world. The team uses test-driven development and continuous integration along with other well established practices that ensure well documented, clean and high quality code.

A large team collaborating on a project brings a number of difficulties when it comes to ensuring software quality. Many developers work on a common codebase and the individual changes have to be put together. A constantly increasing number of team members makes this even more difficult.
Continuous integration aims at resolving the problems of tedious and error prone code merging steps by integrating as early and often as possible. It is a process that requires the entire team's full adherence and painstakingness. Feedback about integration results and the time it takes to receive it is crucial for keeping up motivation.

This thesis introduces the basics of continuous integration, the practices it defines and the tools it requires. The consequences of the increasing volume of the Catrobat project on the continuous integration process are described furthermore. Strategies for mitigating these issues and how these were applied to the Catrobat project are subject to the following sections of

this work. Finally, areas of further improvement are pointed out as future work.

Zusammenfassung

Catrobat ist eine visuelle Programmiersprache, die es Benutzer erlaubt, Programme auf ihren Smartphones oder Tablets zu erstellen und auszuführen. Das Catrobat Team setzt sich sowohl aus zahlreichen Studenten der Technischen Universität Graz zusammen, als auch aus Mitwirkenden auf der ganzen Welt. Das Team arbeitet mit Test-Driven Development und Continuous Integration neben anderen weit etablierten Praktiken, um gut dokumentierten, sauberen und hochqualitativen Code zu gewährleisten.

Der Aspekt eines sehr großen Entwicklerteams, welches zusammen an einem Projekt arbeitet, zieht einige Nebeneffekte im Bezug auf das Aufrechterhalten der Softwarequalität mit sich. Viele Entwickler arbeiten an einer gemeinsamen Code Basis und die jeweiligen Änderungen müssen immer wieder zusammengeführt werden. Eine stetig wachsende Anzahl an Teammitgliedern erschwert diesen Prozess zusätzlich.
Continuous Integration zielt darauf ab, die Probleme des fehleranfälligen und mühsamen Zusammenführens von Source Code zu lösen, indem Code so früh und so oft wie möglich integriert wird. Dieser Prozess erfordert Befolgung und Sorgfältigkeit des gesamten Teams. Ein fundamentaler Aspekt für das Aufrechterhalten der Teammotivation ist das Feedback über die Resultate der Integrationschritte und die Geschwindigkeit, mit der dieses zur Verfügung gestellt wird.

Diese Arbeit stellt Continuous Integration in seinen Grundlagen vor, die Praktiken die dadurch definiert werden und die dazu benötigten Werkzeuge. Die Auswirkungen des wachsenden Umfangs des Catrobat Projektes auf den bereits vorhandenen Continuous Integration Prozess werden im Weiteren beschrieben. Strategien zur Minimierung dieser Auswirkungen und wie diese im Catrobat Projekt angewandt wurden sind Thema der danach folgenden Abschnitte. Zuletzt werden Bereiche vorgestellt, die in zukünftiger Arbeit Verbesserungen erhalten könnten.

Acknowledgements

I would like to thank everybody who supported me while writing this thesis.

Special thanks to Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany for the opportunity and the support throughout my contribution time in the Catrobat organization and to my employer, the Codefluegel GmbH, for supporting and encouraging the progress on this thesis.

Contents

List of Figures

1 Introduction

Continuous integration is a well defined practice that defines several principles that developers have to adhere to. When introducing CI to a project the goal should be that the CI process becomes effortless in order to optimally integrate into the daily routine of every developer.

As the development team of the Catrobat project grew and additional sub projects were introduced, the existing CI process and environment had to be adapted in order to keep CI beneficial to the project. The effects the growth of the project had on the CI process and the strategies used to mitigate the negative impact on it are subject to this thesis.

2 Theoretical background

2.1 Continuous integration

Developing a large scale software project involves multiple developers working on different components of the project. This is due to the fact that in most cases the knowledge about different technologies is distributed among multiple team members. While this is most certainly a good thing seeing that it leads to distribution of responsibility, it introduces certain difficulties to the development process. Independently developed components have to be merged at some point in order to see how (or if) they work together. The integration process becomes increasingly difficult with the number of developers that are working on a project. Multiple different environments and operating systems are used and developers often tend to use outdated or obsolete parts of the code base. This is why the integration process is very likely to reveal problems and should therefore happen as often as possible, or rather continuously.

Continuous integration (CI) is a practice whose aim is to reduce integration problems and improve software quality. Code should be committed to the mainline frequently and the process of checking if a merged version is still building and functional should be automated. Integration has to become a

nonevent [6]. The fundamental features needed for CI are listed below. [6]

- A version control repository containing the source code of every team member
- A build script
- A feedback mechanism such as e-mail, IRC, etc.
- An integration process, which can be carried out manually or automatically by using a CI-server

Figure 2.1 illustrates a typical flow of actions of a CI scenario. Developers are required to commit their code daily. Before every commit, a developer executes a so called private build. This build can contain code inspection tasks as well as test runs among other steps and is meant to ensure that no broken code is committed to the mainline. A CI server is set up to watch the repository and automatically triggers a new build if the polling returned new changes. In case of a broken build the responsible developer has to fix it immediately.

2.1.1 Benefits

Introducing CI to a project requires additional costs at first. This includes hardware/software costs for an integration machine as well as the effort to adapt the development process to meet the requirements of continuous integration. An incremental approach can help converging to a system that yields the main benefits of CI, as described by Paul Duvall in [6].

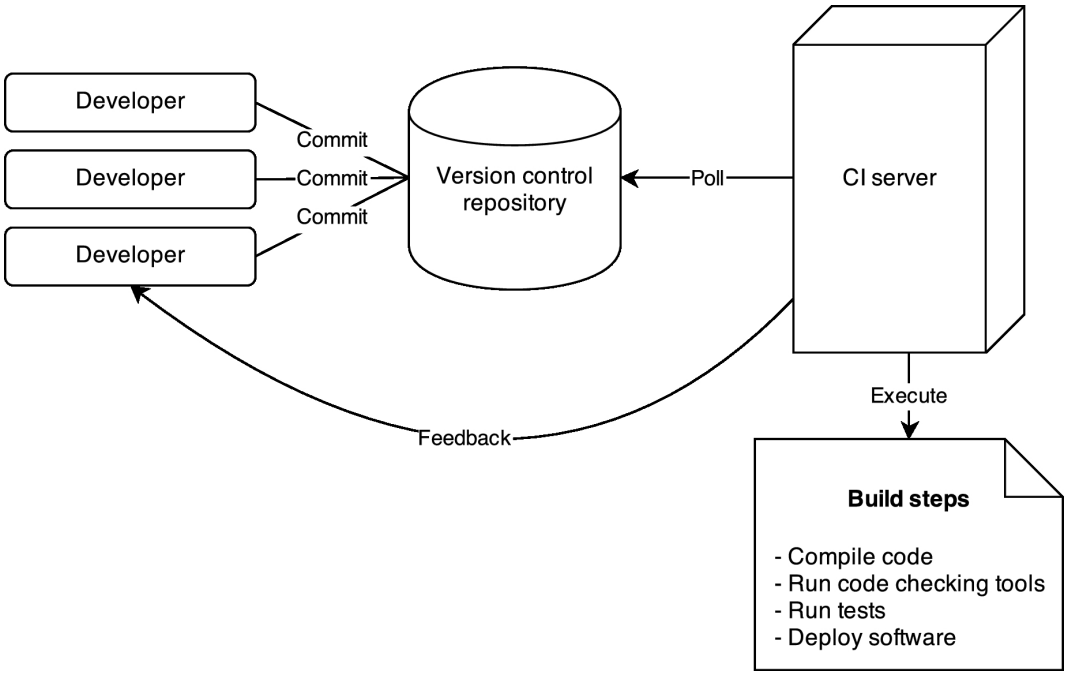

Figure 2.1: CI process.

Risk mitigation

The risk of introducing defects into the code base can be greatly reduced since errors are detected early on. It is the developer's responsibility to fix a build that was broken by changes he committed.

CI also enables measurement of certain attributes like complexity or test coverage which allows tracking the health of the software over time.

Reduce repetitiveness

Software development includes many repetitive processes like compiling code, deploying a database, running test cases or deploying the software itself. These processes require additional time consuming effort and leave room for error if they are carried out manually. Automating the CI process can reduce the repetitiveness of these steps. The integration process is defined to run its steps in a particular order and it always runs the same way. This enables project members to focus on the actual implementation of the software.

Constantly deployable

Every change to the project is constantly integrated to the main version. One of the most important aspects of CI is that errors that occur after the integration process are fixed immediately. This ensures that the main version of a software is in a deployable state at any point. The binaries created by the automated CI server are therefore release-ready at any point in time.

Visibility

The ability to measure certain aspects of a project enables everyone to constantly track the current project state. Every change that is made to the project influences the measurements. The data is gathered constantly and is available at any point in time. Trends can be established and decisions can be supported by the collected data.

Establish confidence

The development team constantly receives feedback on how newly introduced changes influence the existing code base. If code metrics were violated or a bug was introduced that leads to a failing test case, the responsible team member is immediately informed. This can lead to a higher confidence in the product and also reduce the fear of changing code. The trust in the results of the build can also be raised by the fact that the process is carried out the same way every time and is executed on a central instance that is specifically designed for it.

2.2 Continuous integration tools

The following section will go into detail on how continuous integration is carried out in practice. It is meant to give an overview of the parts of a CI system and what purpose they serve.

2.2.1 Build automation

CI requires changes to be merged into the mainline daily. This implies that the code is tested frequently (providing that a private build has been executed before the commit) which further implies a healthy state of the mainline. This state should however not be ensured by private builds alone. An integration machine running a CI build server can serve as a repository monitor. Every change in the master branch should trigger a new build. Should this build fail, it is the committer's responsibility to fix it immediately. This means that a developer's commit is done as soon as the CI server reports a successful build. [7]

Although continuous integration can be practiced manually, repetitive and time consuming tasks that can be automated should be automated. The decision of what to include in the automation is based on what the balance between feedback time and the amount of manual effort should be. A full build may compile, inspect and test the whole project while incremental builds may exclusively compile the project to provide up to date binaries. In all cases the system that performs the automated build must be provided with all necessary resources. This includes build scripts as well as configuration files or third party libraries. Including these resources in the VCS enables a CI build server to stay up to date on configuration or version changes automatically. [7]

2.2.2 Developer workflow

Continuous integration dictates several steps that every developer has to go through multiple times every day [6]:

- Commit code frequently
- Don't commit broken code
- Fix broken builds immediately
- Write automated tests
- All tests and inspections must pass
- Run private builds
- Avoid getting broken code

A team can only fully take advantage of the benefits of CI if it follows these rules.

To frequently (daily) commit code, tasks have to be split up into smaller ones. In theory, running private builds should automatically result in not committing broken code, providing that the private build executes the necessary integration steps and gives suitable feedback. However, broken builds are bound to occur at some point. Immediately fixing a broken build helps keeping the error correction effort smaller and is one of the top priorities when working in a CI environment. A failing test case can be one of the reasons a build is marked as broken. Therefore a solid test suite is of utmost importance for detecting broken code. This is why continuous integration and test-driven development (TDD) are often used in the same context. Section 2.3 describes the practices of TDD in more detail.

A build is only successful if *all* test cases pass. Code coverage tools can help locating weakly tested parts in the software. Code inspection tools can be added to provide additional criteria for build success. To reduce the risk of a broken build, developers should run private builds on their own machines. However, due to environmental differences between these machines build success on the integration machine can not be ensured. In general, members of a team working with CI should avoid working with

code that has been reported to have broken an integration build. These types of defects are likely to spread out and therefore hard to get rid of. It must be safe to assume that the responsible team member is already working on a fix. A good feedback mechanism for the current build status is therefore an important part of CI. It ultimately affects the quality of the software. [6]

2.2.3 Source code repository

Software projects contain a large number of files that are often accessed and changed by multiple developers simultaneously. These files need to be backed up and tracked in order to keep a history of changes and prevent loss of data. Using a version control system (VCS) (or source code manager (SCM), revision control system (RCS), etc.) is without a doubt an essential part of collaborative software development.

A VCS should maintain all necessary files for compiling, testing and building a project so that it is almost effortless to start working on the project after a checkout. [7] This provides a necessary version consistency among all developers and therefore ensures that the build and testing process is carried out in the same way by all team members. Many different tools exist that provide version control. VCSs can be divided into centralized and distributed ones. [1]

Centralized version control system (CVCS) These types of systems (e.g. Subversion[2], CVS[3]) use a single server to store all data that is under version control. The clients fetch the most recent version of the data

[1] Local version control systems are left out here.
[2] `https://subversion.apache.org/`
[3] `http://www.nongnu.org/cvs/`

from a central instance. This is also the main disadvantage of CVCSs. If at some point the version control server is offline, no client is able to commit his local changes to the repository. Failure of the client's local system would cause all changes to his version to be lost.
If the version control server dies the complete version history is lost since it is the only system storing it. [4]

Distributed version control system (DVCS) VCSs like Git[4] or Mercurial[5] allow clients to fetch the complete history of all version controlled data. Every repository checkout provides a complete copy of the repository that it stored on the version control server. In this case if the version control server dies, any client that previously did a checkout from it still keeps the whole file history. This automatically provides a backup mechanism that gets more and more redundant with an increasing number of clients. [4]

Version control systems can be used with different workflows. Distributed VCSs provide excellent capabilities for branching and merging.

Feature branches are a widespread concept used in distributed version control systems. Developers branch away from the mainline every time they start working on a new feature. As soon as the feature is completed the branch is merged back to the mainline. Changes that happen in the mainline will be merged into the individual feature branches throughout the development of that feature. The point in time this merge happens can be determined by the feature developer himself although the sooner this merge is carried out the less tedious it is. One of the main advantages this model provides is that individual features can easily be excluded from release by not integrating them into the mainline. However this leads to a longer

[4] `http://git-scm.com/`
[5] `http://mercurial.selenic.com/`

lifetime of the individual branches.

Feature branching is a controversial topic when talking about continuous integration.

Feature branches vs. continuous integration

"Now there may be valid reasons why you're doing feature branching [...] but you can't say you're doing continuous integration and be doing feature branching. It's just not possible." [15]

Jez Humble takes a very clear standpoint on feature branching and continuous integration. His statement is supported by Martin Fowlers article about the topic. [9]

Figure 2.2 illustrates a basic feature branching model. According to Fowler [9] this breaks the continuous integration concept.

By definition of Fowler continuous integration requires all changes to be merged into the mainline frequently, usually at least daily. [7] The basic feature branching model isolates developers from each other. If a large feature is merged into the mainline, all open feature branches have to incorporate the changes which can lead to a large amount of conflicts (G1-6 in Figure 2.2). Additionally with this approach the individual features are never tested together until all of them have been integrated to the mainline. Feature branches tend to live longer than one day and if the merge into the mainline happens at the very end of the feature development the integration is not considered to be continuous.

A reworked version of this model is shown in figure 2.3. As soon as a change has been made in a feature branch it is immediately integrated into the mainline. This approach mitigates the additional effort that comes with large merges and provides faster feedback for the developers on how their feature performs with the others. It is still worth noting that unless feature branches last less than one day this practice is not considered to be continuous integration by definition of Martin Fowler. Integrating multiple times a day means that features have to be split into several small tasks which encourages planning out the development process very carefully. The main advantage of feature branches mentioned earlier (keeping features from release) can still be kept by using feature toggles. [9] These can be used to hide unfinished parts of a feature from the user.

Distributed version control systems can be used for continuous integration when using a suitable branching model. There has to be one branch that is considered to be the mainline where changes in other branches are pushed to continuously.

In a centralized version control system a broken build can potentially halt development. If a developer breaks the build he has to fix it immediately. During this process no one is allowed to check out the mainline or commit to it. When using a DVCS each developer can pull the mainline into his branch. The integration build can then be executed on the developer's branch. This way a broken build does not affect the mainline. As soon as the build is successful the changes can be pushed back to the mainline without any conflicts (providing that no changes have been pushed to it in the meantime). The choice of which version control system to use to implement CI very much depends on the experience, expertise and, most of all, discipline of the development team and the nature of the project. Open source projects often use DVCSs because they make it easy for external developers to contribute to the project. Since access to the main repository

is often restricted to internal developers systems like GitHub[6] allow forking the original repository to enable others to work on the project. Changes they made are integrated back to the main repository by using pull requests that can only be accepted by specifically assigned members of the original repository.

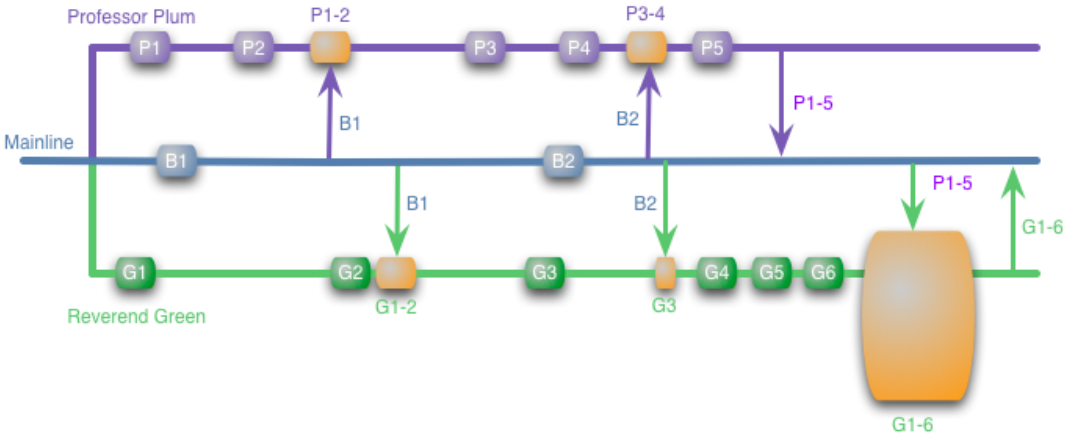

Figure 2.2: Basic feature branching [9]

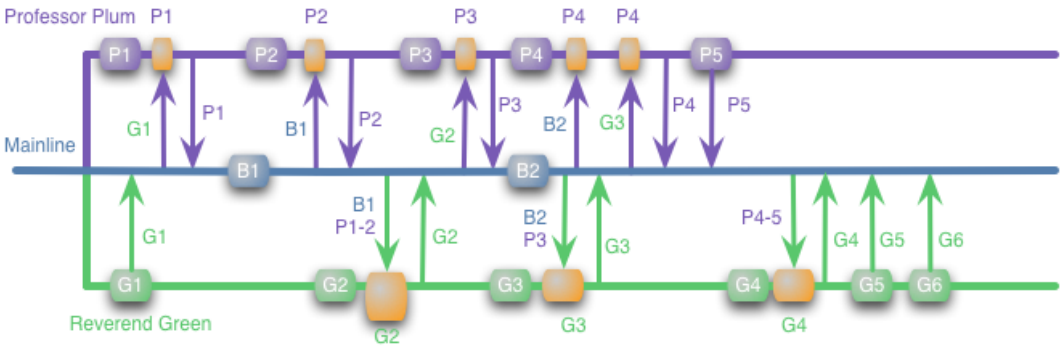

Figure 2.3: Feature branching with CI [9]

[6] https://github.com/

2.3 Test-driven development

Arguably, continuous integration can be practiced entirely without any testing process. A build may only compile the code and check for coding standard conformance. But practicing continuous integration without automated tests will not utilize the available infrastructure to its full potential. [28] The correct behavior of a software product can hardly be verified by simply compiling it. An effective test suite is an important aspect for successfully ensuring software quality.

Test-driven development can help building up a high quality test suite. A test case is written before the code it should execute even exists. With this approach developers are forced to write testable code and encouraged to write this code as simple as possible. Refactoring the code is the third step of the so called "heartbeat" of TDD as described by Lasse Koskela [23].

2.3.1 Test-code-refactor

"Only ever write code to fix a failing test case."[23]

In other words this means that a test case is always written first and the code that makes it pass is written afterwards. This is contrary to the common approach that software development starts with a design process and after a suitable design is established developers would start to implement it. Test-driven development puts the designing process at the end of the development cycle as illustrated in figure 2.3. The designing process is called refactoring since it differs from the traditional designing approach. [23]

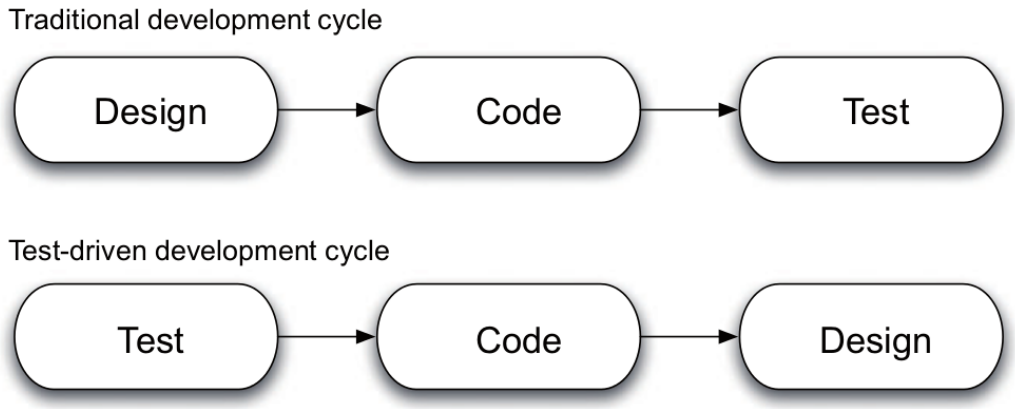

Figure 2.4: Traditional versus test-driven development cycle [23]

Writing a test case before the actual code exists guides the thought process into a more usage-oriented one. Developers have to think about how the code should be used and document this with a test case. In a way the resulting set of tests can be seen as a documentation of requirements that have to be fulfilled. The code that is written to fulfill these (i.e. pass the test cases) is in its first iteration simple and just good enough to turn the test case from red to green.[7] The code may be simple and suboptimal since it is revisited in the last step of the TDD cycle - the refactoring process. Refactoring is meant to ensure that the design aspect of software development is not left behind. The effort of maintaining poorly designed code makes the importance of this step obvious. Refactoring includes the production code as well as the corresponding test code.

[7]Most development environments use red/green to mark fail/success results of a test run.

2.4 CI anti-patterns

Practicing CI for some time can yield the effects of anti-patterns that have unknowingly been introduced while trying to introduce the practice of CI itself. Paul Duvall describes a number of these patterns in [5] that are further discussed below.

2.4.1 Infrequent check-ins

By keeping code out of the repository the integration process is delayed which increases the difficulty of it. When using CI commits should occur at least daily to ensure that builds are triggered frequently and feedback about the integration is received regularly. One of the reasons for code not being committed to the mainline is that tasks are too big to finish in one day or less. To keep up the frequency of commits tasks have to be divided into smaller ones. At every commit of such a small task developers immediately get feedback of how this change is working with the rest of the code. [5]

2.4.2 Broken builds

Builds should be allowed to break at some point since this is a necessary notification of a software defect. The longer a build stays broken though, the harder it is to fix the defect. As mentioned before it should be the responsible developer's top priority to fix the issues causing the failure immediately after receiving feedback about it. Builds that stay broken for a long time keep developers from checking out functioning code. Carrying out a private

build before every commit to the mainline can help preventing broken builds. A developer would check out the latest version of the mainline and make his changes locally. After finishing a task (that is small enough to keep up the integration frequency) he integrates the mainline version again and runs a local build, which immediately yields feedback about the state of the software. Only after this private build passed the changes are committed to the remote repository. This can help reducing the occurrences of broken builds on the integration server. [5]

2.4.3 Minimal feedback vs spam feedback

Feedback about the build status is one of the most important aspects of CI. Receiving no feedback at all keeps anyone from knowing when changes broke the build. On the other hand receiving an e-mail about every build status at any time may lead to team members ignoring the messages. The feedback mechanism may vary between different projects. If all developers share an office visual feedback in the form of large screens displaying the build status or light bulbs that reflect it may prove as most effective. However, projects where team members are more distributed (e.g. open source projects) an e-mail can be more useful. These should however be targeted at specific team members (project lead, members that recently committed changes, ...) to reduce the feeling of being spammed with feedback messages. [5]

2.4.4 Slow build server

A state of the art machine used as build server is highly beneficial for reducing the feedback delay. While the acquirement of it might introduce additional costs, these are easily compensated by the fact that the increased speed of the build execution saves the team time and therefore money. The sooner an integration is reported to be successful, the sooner developers can begin to start their next tasks. [5]

2.4.5 Bloated build

A common anti-pattern that comes with the ease of automation that build servers provide is to include as much work as possible into the build. This results in a substantial increase of the execution time and therefore an increased delay of feedback. As a first step to mitigate these issues the execution time of the individual build steps (compilation, test cases, etc.) could be decreased. Upon reaching the cap of build step optimization, a large performance increase can be reached by introducing build pipelines. A commit would trigger a small build that may only compile the code and execute a fast set of unit tests and that upon success triggers a successive build that may execute more time consuming tasks. The first build however would provide fast intermediate feedback about the basic health of the project. [5]

2.5 CI in open source projects

Open source projects are often driven by distributed teams with contributors around the world. While the core CI process and technology stays the same for any kind of project, the distributed nature of the development team introduces certain difficulties. [16]

External developers that contribute to open source projects are often volunteers that are not paid for their work. Since these developers basically are not bound to any rules or time constraints it can be very difficult for the core team to make certain practices mandatory. Geographical distribution makes this additionally harder. [14] Open source projects should therefore find the right mechanisms and tools to enable every contributor to easily begin practicing continuous integration. Communication platforms such as IRC or Skype should be available at any time and for everyone. Breaking the barriers of communication is important to establish trust within the team and can be achieved by including external developers in meetings using videoconferencing tools.

Open source projects should not have to adapt the CI process but rather practice it with increased accurateness for it to yield the benefits it can provide. [16]

2.6 Scaling continuous integration

The success of continuous integration greatly depends on the discipline of the team. Therefore the environment that is provided should make the

process as easy and effortless as possible to keep up the teams motivation. With an increasing size of a project the continuous integration process has to scale appropriately. The following sections will further discuss the main issues that can be experienced by the scaling process according to [26].

2.6.1 Increasing size of code

When practicing TDD an increasingly large code base implies that more and more test cases are added to the test suite. This results in longer build times and slower integration feedback. To avoid continuing to work on a broken code base developers should wait until the integration build returns its results. A long waiting period results in an extension of unproductive time. Developers tend to commit less often to keep these periods at a minimum. However the less frequently these commits happen the more changes they include and the larger (and possibly harder) the integration steps become. Merge conflicts become more likely and fixing broken builds also becomes increasingly difficult.

The time it takes for one developer to receive feedback on his latest commit is also the time other developers are kept from integrating their changes into the mainline. If developers leave work before the integration build is finished and returned its results, the remaining team members are forced to fix a possibly broken build since otherwise they can not commit their changes to the mainline by definition. A workaround for this issue might be to commit changes at the beginning of the day to leave enough time for fixing any occurring issues. This however implies that the commit is ready at that point since otherwise it has to be postponed to the next day. Starting to work on other issues in the meantime again results in larger changesets and increasing likeliness of failure.

All these aspects cause the development process to drift away from continuous integration. [26]

2.6.2 Increasing number of developers

The speed at which the code size increases is of course tightly coupled with the number of developers that are working on a project. This aspect has been mentioned in the previous section. But an increasing number of people added to the CI process also means that more and more people depend on green builds. It is very likely that one developer wants to commit his changes while another one is currently working on a fix for his broken build. During this time no one can commit to the mainline by CI convention (that is if they are not working on the fix as well). This situation shows the importance of this convention since another commit to the mainline would further increase the difficulty of fixing the defect as well as the time the build remains in a broken state. However since every team member aims to finish their work it is likely that their commit will happen irregardless of the current state of the mainline. This leads to a situation in which no team member feels responsible for fixing a broken build since it can not be known for sure whether or not the individual changes would have passed the build process. The longer the build stays broken the lower the inhibition threshold for committing changes to the already broken mainline.

Although developers should never check out broken code, an increasing lifetime of this state will result in it happening anyway. Therefore the failures that were introduced in the mainline version spread out to the individual workspaces of each developer. In the worst case scenario this means that each of them has to individually fix the same issues which is unnecessarily redundant.

Continuous integration requires team members to run private build before committing any changes to ensure that no broken code is introduced into the mainline. Pulling broken code from the mainline implies that the failures at hand have to be fixed before any changes can be committed. This slows down the integration process dramatically and discourages developers from committing frequently which is obviously detrimental to the CI process. [26]

2.7 Scaling strategies

The aspects described above have been experienced as the main reasons for a development process to drift away from a continuous integration approach. As the team grows the CI environment has to grow with it to ensure that every team member is highly encouraged to practice all the necessary steps that define CI. This includes fast feedback as well as nearly effortless automated build support. Experience has shown that increasing integration effort results in less frequent commits and therefore fewer build and test runs. Continuous integration is meant to simplify the integration process by dividing it into smaller incremental steps. The following sections discuss several strategies for scaling the process as described in [26].

2.7.1 Limiting the build time

The execution time of the integration build is closely related to the frequency of integration. The longer the build takes, the less frequently developers will be integrating their changes. To determine a reasonable build time

the desired relation between it and the integration frequency has to be established. If developers have to wait 1 to 10 minutes for the build to finish they can be expected to integrate every hour. However if the build time starts to exceed this threshold this expectation becomes unreasonable. The time it takes for a developer to integrate his code into the mainline should be relatable to the time it took him to implement it otherwise the process loses its benefit of becoming a non-event.

Keeping the build time low can be a difficult task to achieve for larger projects with large test suites so the tolerated balance between build time and integration frequency is to be determined. [26]

2.7.2 Separate the build process

Sequential builds are poorly scalable with a growing number of steps they have to execute. The automation aspect of the build process may encourage to add as much work as possible to it. This results in rapidly increasing execution times. A more versatile approach is to separate the build into smaller independent builds that can be executed simultaneously. This helps reducing the build time as well as prioritizing certain steps in the integration process. The goal is to fail fast meaning that feedback about code that is not compiling or a failing test case should be provided as early as possible.

Additionally there should be multiple different builds for different purposes. These can be established by analyzing the needs of different teams in the project. Developers are most likely more interested in receiving feedback about compilation or test results while the latest executables may be more interesting for the marketing department. Handling different purposes with different builds can greatly decrease the delay of continuous feedback and

ultimately also increase the integration frequency. [26]

2.7.3 Faster unit tests

Effectively working with test-driven development requires a certain amount of experience. Tests have to be clearly divided into unit and functional tests to avoid redundancy. Unit tests should in general run very quickly and avoid leaving the function or object they are supposed to test. A poorly designed test suite can be the main reason for long build times. To enable the segmentation of the build process and execute multiple parallel builds the individual tests have to be independent. As mentioned before, the refactoring step in the TDD cycle also applies to test cases. CI servers provide great capabilities for measuring execution times and make it easy to find slow test cases. These should be continuously refactored towards a lower execution time. [26]

2.7.4 Small teams

The main disadvantage of large teams when practicing continuous integration is the number of people that are affected by a broken build. In the real world breaking a build is inevitable. Therefore it has to be guaranteed that a broken build does not hold up the development process of the whole team. By dividing into multiple smaller teams that are assigned to different fragments of the project a certain amount of independency can be introduced. These teams run individual builds that only test their part of the code. If a commit to the mainline breaks the build it only affects members of the team that is assigned to the broken module. However to allow this

separation the code has to be properly separated into several modules that can be implemented and tested independently. [26]

2.8 Android

This section provides an overview over the Android platform and is meant to give a better understanding of application development for the OS and the testing process.

2.8.1 The multitude of devices

Android is an open source operating system that is managed by a group of 84 [1] organizations forming the "Open Handset Alliance"[8]. The decentralized nature of control entails certain aspects that increase the difficulty of application development. [30]

Multiple screen sizes Devices running Android come with a large number of different screen sizes.

Fragmentation Due to the multitude of device manufacturers and their freedom to choose different versions of Android, applications have to support multiple versions of the OS.

Difference in hardware It can never be assumed that a specific hardware feature is supported by the device an application is installed on. (Sensors, GPS, ...)

[8] `http://www.openhandsetalliance.com/`

Resource limitations The amount of available resources differ between devices and are in general very limited.

The diversity of Android devices is illustrated by figure 2.5 which shows all devices that downloaded the OpenSignal application in the months before July 2013. [19]

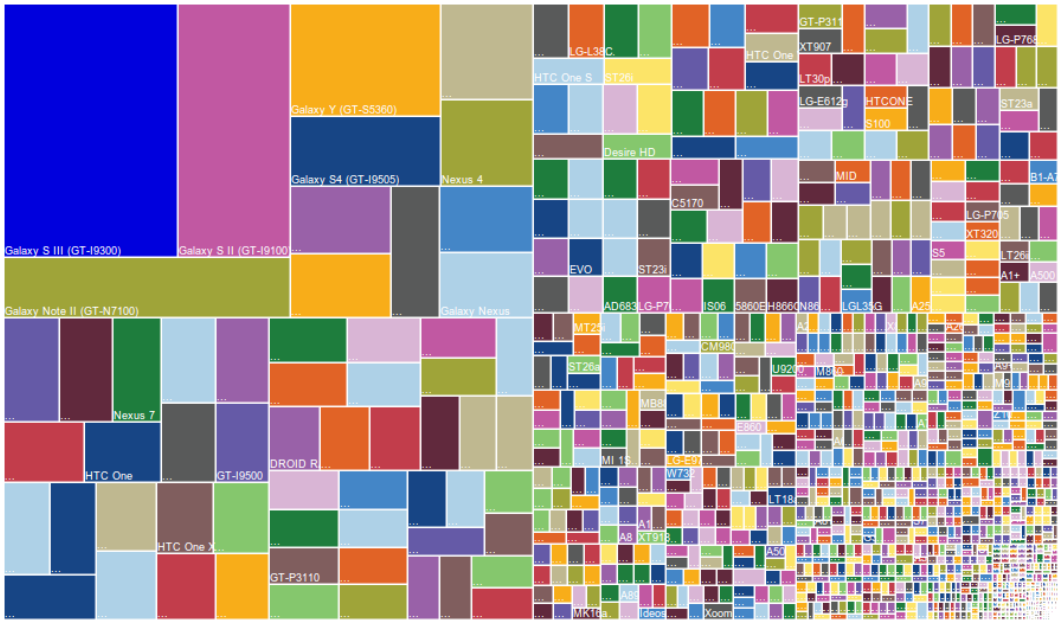

Figure 2.5: Android fragmentation [19]

2.8.2 Build process

The first steps of the Android build process illustrated in figure 2.6 resemble the standard Java build process with the addition of generating a resource file and Java interfaces from *.aidl* files. The compiled Java classes are converted to *.dex* format that is compatible with the Dalvik virtual machine

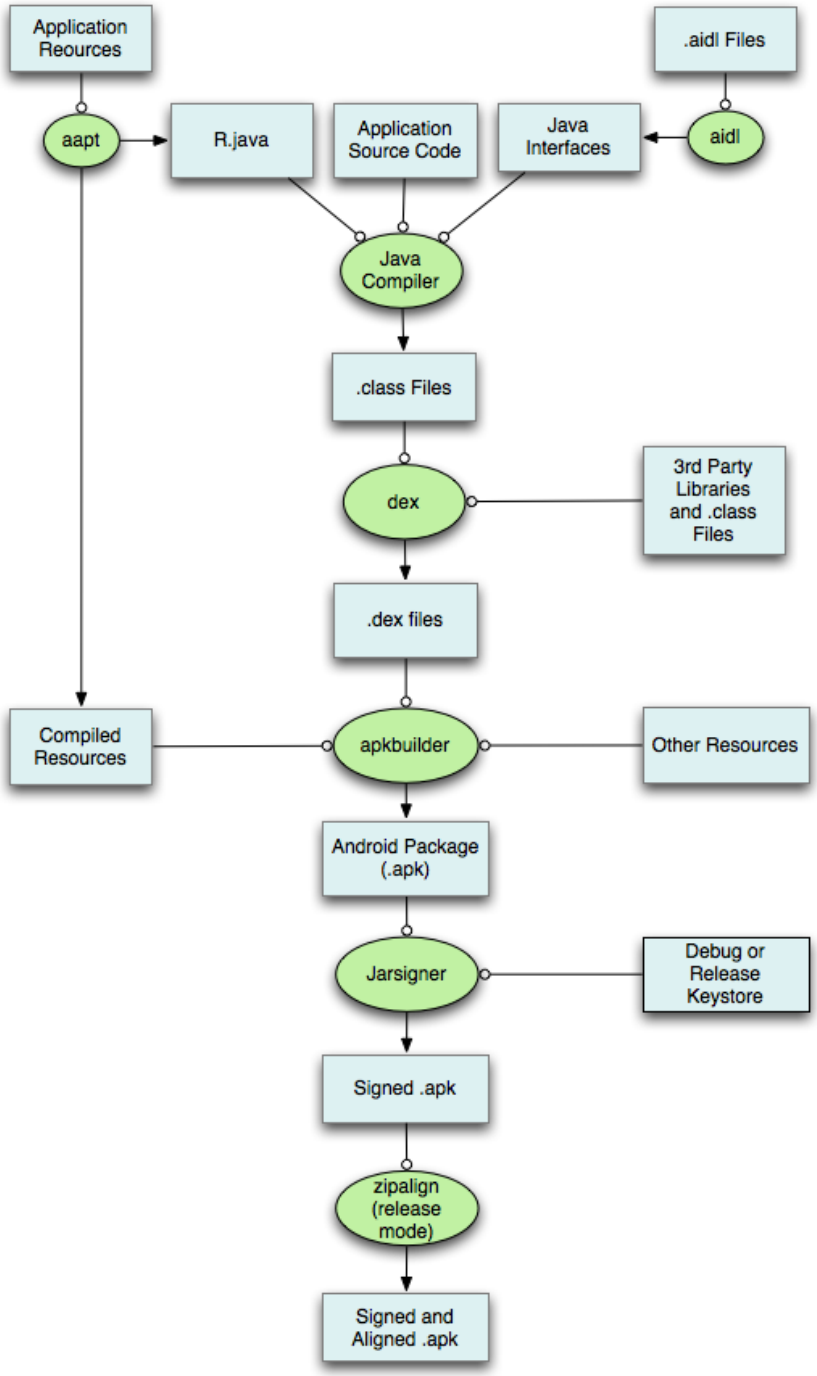

Figure 2.6: Android build process [11]

running on Android devices. All compiled and non-compiled resources are packaged into an *.apk* file that is used to install an application on an Android device or emulator. [11]

The Android SDK provides an Ant[9] build script that is used to build an application from the command line which enables automation of the process. Keeping this build script under version control allows keeping track of changes that come with SDK version updates or manual modifications to it (e.g.: additional build targets).

2.8.3 Application lifecycle

An Android application can include multiple so called *activities*. These activities hold UI elements such as buttons, list views etc. and handle user events. Every activity has its own lifecycle as illustrated in figure 2.7. Resources that have been allocated during the creation of an activity and are exclusively needed by this activity have to be correctly released when it is sent to the background. This event can be triggered by either the user himself or external events such as receiving a phone call or a display timeout. The introduction of fragments further increases the complexity of these lifecycle events. Fragments are reusable UI parts that can be used to assemble multi-pane activities. [12] Figure 2.8 shows a simplified diagram of the lifecycle of a fragment. Correctly handling each of these lifecycle events is crucial for correct application behavior. There are a number of frameworks that provide the capabilities for testing Android activities and fragments natively, one of them being Robotium. [10]

[9] `http://ant.apache.org/`

[10] `https://code.google.com/p/robotium/`

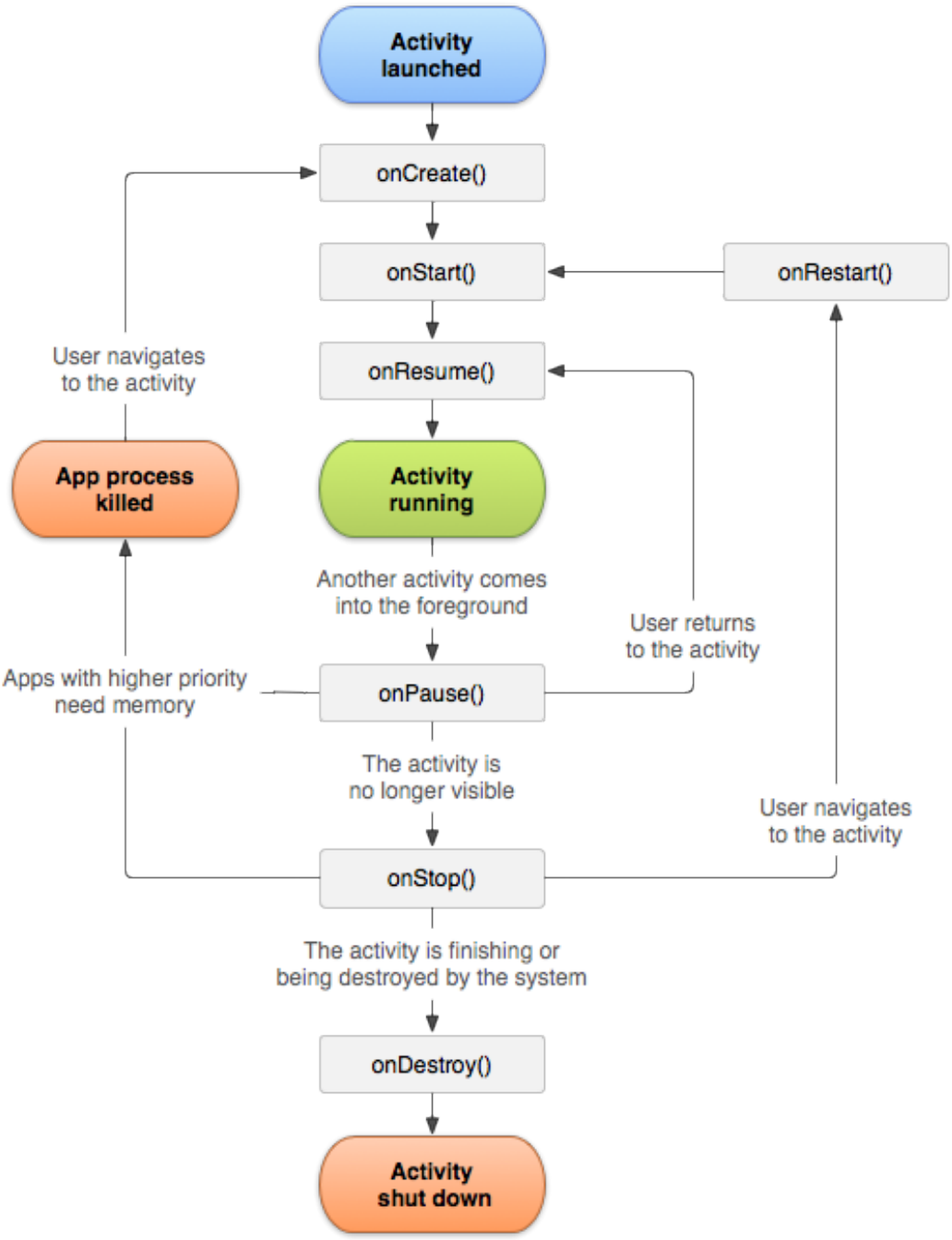

Figure 2.7: Activity lifecycle [10]

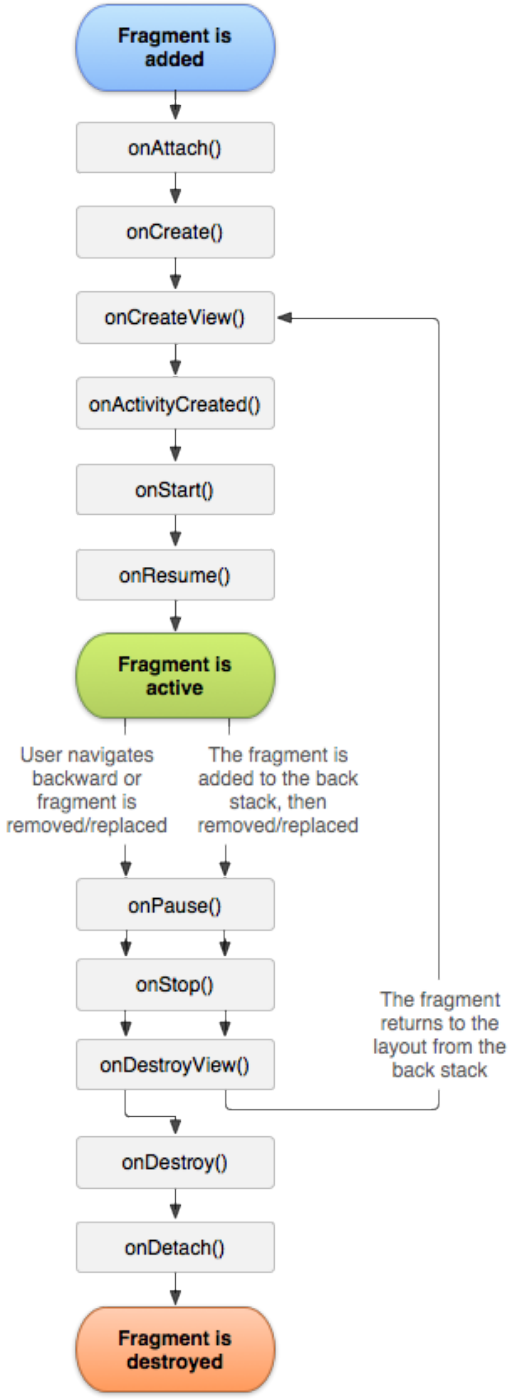

Figure 2.8: Fragment lifecycle [12]

2.8.4 Android application testing

Android applications include platform specific code as well as platform-independent Java code. A test suite for Android can therefore include test sets that can be executed without installing and running the application on a device or an emulator providing that non-Android-specific code is properly encapsulated. Test cases for Android specific code however need to be executed with the application under test being installed and running on an emulator or a real device. It is important to consider the pros and cons of testing in a simulated environment versus testing on actual smartphones or tablets.

One of the biggest advantages of emulated Android devices over actual hardware is cost efficiency. Given the before mentioned fragmentation of the Android ecosystem it is unfeasible to cover as many different devices as possible by acquiring every one of them. Besides the cost of the machine a virtual device is running on, Android emulators are free and can be configured to simulate any version of the mobile operating system. This configuration process can be automated which makes it easy to add to the integration routine on a CI build server. The most obvious disadvantage of emulators over real hardware is that a simulation is never 100% representative of the real world. Simulation errors can produce entirely different results on a virtual device. This can lead to a number of false negative errors that can not be reproduced on real devices. Due to the open source nature of the Android operating system, smartphone and tablet manufacturers are free to customize the experience on their products to a great extent. These customizations are not reflected in an emulated Android device which can lead to unexpected behavior when it comes to testing user experience. [29] Additional limitations of the Android emulator include lack of Bluetooth support, USB connections or SD card insert/eject events among others. [13]

These differences show that none of the two is the best solution for all testing phases. Choosing the right method for the right phase is an important aspect to consider. [29]

When practicing continuous integration it is required that there exists a central integration machine where the build process is carried out the same way on every execution. This way developers can be sure that their changes did not break the working state of the software. In an Android project the integration machine should provide a real device in addition to automatically configured Emulator instances that are used for running the integration builds. This way it can be guaranteed that all developer share the same testing environment for their integration steps.

The before mentioned application lifecycle in Android encourages the use of UI tests to accurately verify the application's behavior. Due to the latency of UI testing frameworks and the device UI itself the execution time of these tests is significantly higher than regular unit tests. Including UI tests in the integration test suite will increase the execution time of the build and therefore delay feedback. It is important to consider how to integrate these tests in the automation process.

The challenges of mobile application testing in general are not limited to the choice of the testing environment. Multiple sources [22] [27] [21] describe different areas that are characteristic to mobile application development and that pose new challenges to the development and testing process, some of which are mentioned below.

Connectivity Mobile devices are logged in to the mobile network at all times. The speed and reliability of this connection varies depending on network provider or location. Network access may also be limited

when accessing public Wi-Fi networks. Testing an application in different connectivity scenarios is therefore important to mitigate errors in these situations. [22]

User interface As mentioned before the aspect ratio and physical size of mobile devices varies between manufacturers and models. This leads to user interfaces appearing differently on different devices. Covering as many of them as possible when testing an application is therefore recommended. [27]

Resource constraints While the overall performance of mobile devices is rapidly increasing the gap between mobile and desktop hardware is still a large one. Resource usage has to be taken into account to avoid performance issues. [22]

Context awareness Sensors (motion, noise, light etc.) and different types of connections (Bluetooth, Wi-Fi, 3G etc.) provide a large amount of data that is commonly used by mobile applications. Different devices may have differently calibrated sensors and connectivity characteristics vary constantly. It is a challenging task to test whether an application performs correctly under different circumstances. [22]

Security The challenge of protecting privacy and ensuring security is tightly coupled with before mentioned aspects. The variation of security levels throughout different Wi-Fi networks is an important aspect to consider. Considering the availability of a large amount of contextual and personal information on a mobile device, security issues pose a substantial threat to privacy. [22]

3 Practical Part

3.1 Catrobat

Catrobat is a visual programming language for smartphones, tablets and mobile browsers. The Catrobat umbrella project houses several projects including the Android distribution of the IDE for the creation of Catrobat programs called Pocket Code, an Android image editor named Pocket Paint, the community website and distributions of the individual applications for different platforms among others. It is a student-driven open source project with a large amount of contributors that practices test-driven development and continuous integration.

3.1.1 Development process

While the core team resides at Graz University of Technology, the Catrobat project has contributors around the world. This aspect introduces certain difficulties when it comes to practicing continuous integration.

Catrobat uses GitHub[1] for version control. To a certain amount the access to the version control system has to be restricted. Public write access would make it impossible to control what is introduced to the project. Therefore only the core team is capable of committing changes to the mainline of the project. To enable non-core members to contribute to the project, Catrobat uses pull requests. This system has been widely adopted by many different open source projects. Developers issue such a request to notify core members that they want their changes to be accepted. Upon accepting a pull request all changes it contains are merged into the mainline. This however is only possible if the merge can be carried out automatically. Therefore all conflicts have to be resolved by integrating the mainline into the developers branch prior to issuing the request.

The GitHub pull request system makes the changes a pull request would introduce very transparent and easy to review. This is why the core team has adopted this practice as well. In the Pocket Code project integration into the mainline is done exclusively via pull requests. Although the pull requests are reviewed carefully there can be no guarantee that the changes that it introduces will not break the functionality of the mainline version. To verify the success of the integration process a build pipeline has been instituted that will trigger on every pull request. The environment that has been established to enable this process will be discussed in the following sections.

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

3.2 Scaling CI in Catrobat

The strategies described in section 2.7 for scaling continuous integration all revolve around one main goal - reducing the time it takes to get feedback about the result of the integration process. Fast feedback is the most important aspect for keeping up the motivation and therefore the integration frequency of all team members. The first instance of the continuous integration environment in Catrobat was set up to deal with few jobs and relatively short execution times. The CI build server was (and is still) running Jenkins[2]. The machine running the instance of Jenkins had one single executor for build jobs. This was due to the fact that the server was running Android build jobs exclusively. As mentioned before, Android test cases have to be executed on an emulator or a real device. Exclusively testing the Android applications on emulated devices was not an option since OpenGL[3] support is required which the Android emulator lacked at the time. A real device was therefore attached to the CI build server via USB. If one job started, the CI servers executor was reserved for it and all subsequent jobs had to wait in a job queue. Multiple executors would have caused the queued jobs to start while the Android device was still in use by another job. The first job would fail because the second would have started executing tests on it thereby interfering with the first testing process.

The job queue caused feedback times to rise constantly. At the same time the number of test cases increased rapidly which caused the job execution time itself to increase. When practicing continuous integration slow builds and late feedback are detrimental to the development process and ultimately to the quality of the software.

[2] http://jenkins-ci.org/
[3] https://www.opengl.org/

As mentioned before, the CI environment of a project has to scale with the number of team members and the increasing size of the code base. Since all of Catrobat's projects are open source and have a large amount of contributors that is constantly increasing, the existing environment had to be adapted to provide sufficient capabilities for practicing CI. The following sections describe the tools and practices that were used to ensure these capabilities and at the same time provide an environment that scales with the future growth of the Catrobat project.

3.2.1 CI server

In its basic setup the build server includes a single machine running the main Jenkins instance, the master node. Several build jobs can be configured through the web interface this node provides and the individual workspaces of these jobs are maintained by the Jenkins master. All build results such as test results, log files or executable binaries are managed by this node.

The master node can manage several additional slave nodes that are used to execute build jobs. These nodes can be instantiated on the same machine as the master itself or on a remote machine that can be accessed via SSH[4]. A slave node is represented by a running instance of Jenkins' slave.jar file.

It has been mentioned before that different tasks should be handled by different builds. These builds most likely require different resources that have to be managed properly. One of the main advantages of the master-slave system that Jenkins provides is that resources that are needed for building and testing a project can be distributed and exclusively reserved for specific builds. This is done by labeling the individual slave nodes

[4] http://en.wikipedia.org/wiki/Secure_Shell

according to their capabilities. Figure 3.1 illustrates an example of a master slave setup with server 1 and 2 being two separate physical machines.

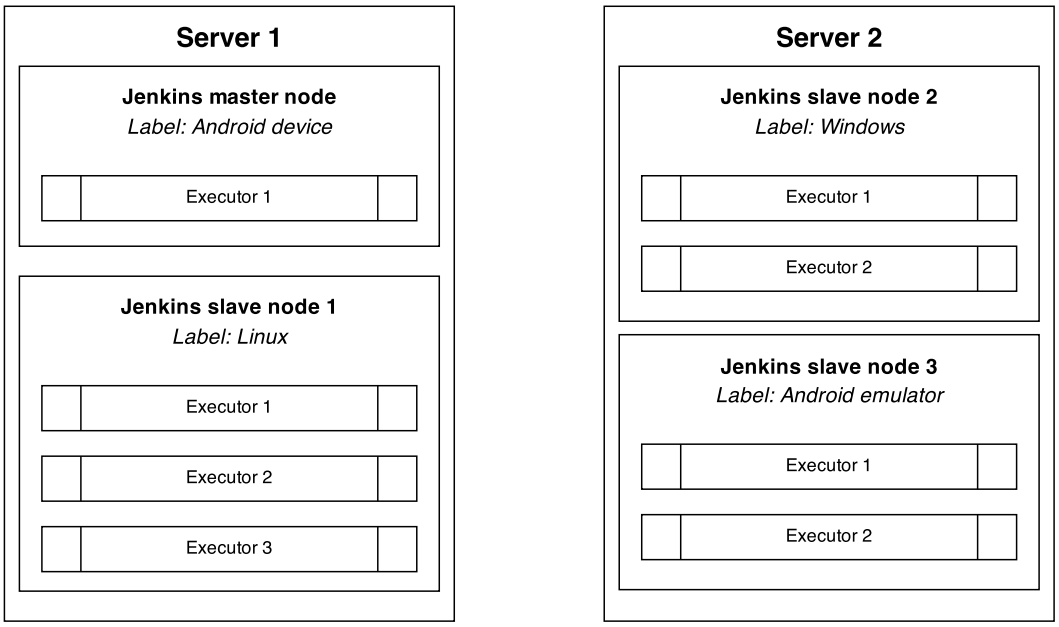

Figure 3.1: An example for a master slave setup

The master node may have an Android device attached to it and may therefore receive the label "Android device". All jobs that execute tasks that require a device are assigned to execute exclusively on nodes that are labeled accordingly. Jobs that don't need any specific resources for execution are assigned to run on any other node but the ones labeled with "Android device". These jobs would otherwise occupy resources they don't need and block the ones that do from execution which again can greatly increase feedback delay.

In addition to labels every node can have multiple executors that can each run a single build job. This enables one node to execute multiple jobs in

parallel. For this to be possible the individual jobs have to be independent from each other, meaning that the tasks they execute have to be clearly separated. A test suite that can be split into several smaller sets is therefore an important prerequisite.

The individual nodes that the Jenkins master manages should share a basic configuration including the set of installed packages and dependencies. As these are most certainly subject to change as new versions are provided, it is important to keep all nodes up to date. Build jobs are configured once and if not assigned to a specific label they can be executed on any node. Therefore it has to be guaranteed that the required basic environment is provided on all nodes. To handle this problem a set of scripts can be provided that can be executed by any build job before the actual work is done. Keeping this set under version control ensures that it is up to date on every node and using Jenkins' per-node environment variables can guarantee that all scripts can be executed on all nodes.

This setup now provides a platform where the individual sub teams of the Catrobat project do not block each other from receiving fast feedback about their integration process. Projects now have exclusive nodes that execute their CI builds which minimizes the time to wait in queue.

3.2.2 Build pipeline

The integration cycle in the Catrobat project begins with issuing a pull request. This automatically triggers a build job on the Jenkins CI server that executes a set of fast unit tests and code inspections. The result of this build is directly connected to the state of the pull request, meaning that a failure will automatically mark the request accordingly. This prevents core

members from accepting requests that in fact would break the state of the mainline. While developers are still required to execute private builds prior to committing their changes, this step provides additional verification before failures are introduced. Not only does it verify the basic functionality of the software, it is also a helpful tool to check if developers adhere to the basic concepts of continuous integration. Figure 3.2 shows a pull request that has been marked by a successful build. This figure also includes Catrobats version of the *Integrate* button ("Merge pull request") that Paul Duvall mentions in [6]. The Jenkins CI server is set up to watch the mainline of

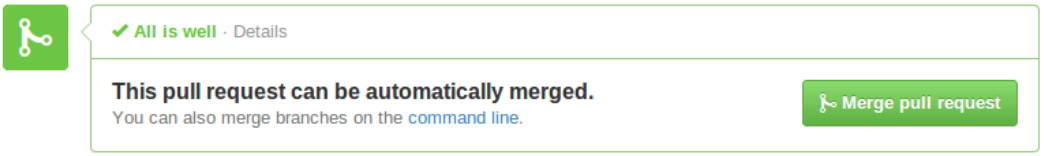

Figure 3.2: A pull request marked by a successful build [17]

the version control system for any changes. This means that upon accepting a pull request by clicking the button illustrated in Figure 3.2 the server automatically triggers a build job that executes an integration build. In Pocket Codes case this job includes code inspections, fast JUnit tests and a large set of functional Android UI tests. Since this job represented the largest overhead in Catrobats CI process the optimization of it was considered to yield the largest improvement in terms of feedback delay. By separating the existing sequential build into multiple stages the overall execution time was significantly decreased. The initial stage provides fast feedback by executing fast Unit tests while functional UI tests are pushed further back in the pipeline. While this improves the performance of the feedback process it also represents first steps towards a deployment pipeline as described by Martin Fowler in [8] and in more detail by Jez Humble in [16].

The first stage of the build pipeline should include a large enough test set to accurately verify that the integration was successful. If later stages that execute functional and regression tests point out further bugs in the software this information can be used to further extend the Unit tests in the first stage of the build. Nevertheless all stages have to be treated with equal care meaning that a build should be fixed immediately regardless of the stage that caused it to fail.

Figure 3.3 illustrates the build process of Pocket Code after the acceptance of a pull request. The separation of the build process into multiple stages required the adaption of the test environment which will be described in the following section.

3.2.3 Test environment

Pocket Code provides functionality that requires additional hardware such as robots that can be controlled via Bluetooth or Arduino[5] circuit boards. This functionality is also tested with real hardware. Since the Android emulator does not support Bluetooth connections these tests have to be executed on actual Android devices. To be able to fully take advantage of the parallel execution capabilities of the Jenkins master-slave system the test suite for Pocket Code had to be separated into multiple smaller ones that can be executed simultaneously. This was achieved by distributing the application binary to multiple emulators and hardware devices that each execute a predetermined test set. The individual sets are independent and tests that require a hardware device for successful execution are marked using Java annotations[6]. These annotated test cases are automatically selected and

[5] http://www.arduino.cc/

[6] http://docs.oracle.com/javase/tutorial/java/annotations/

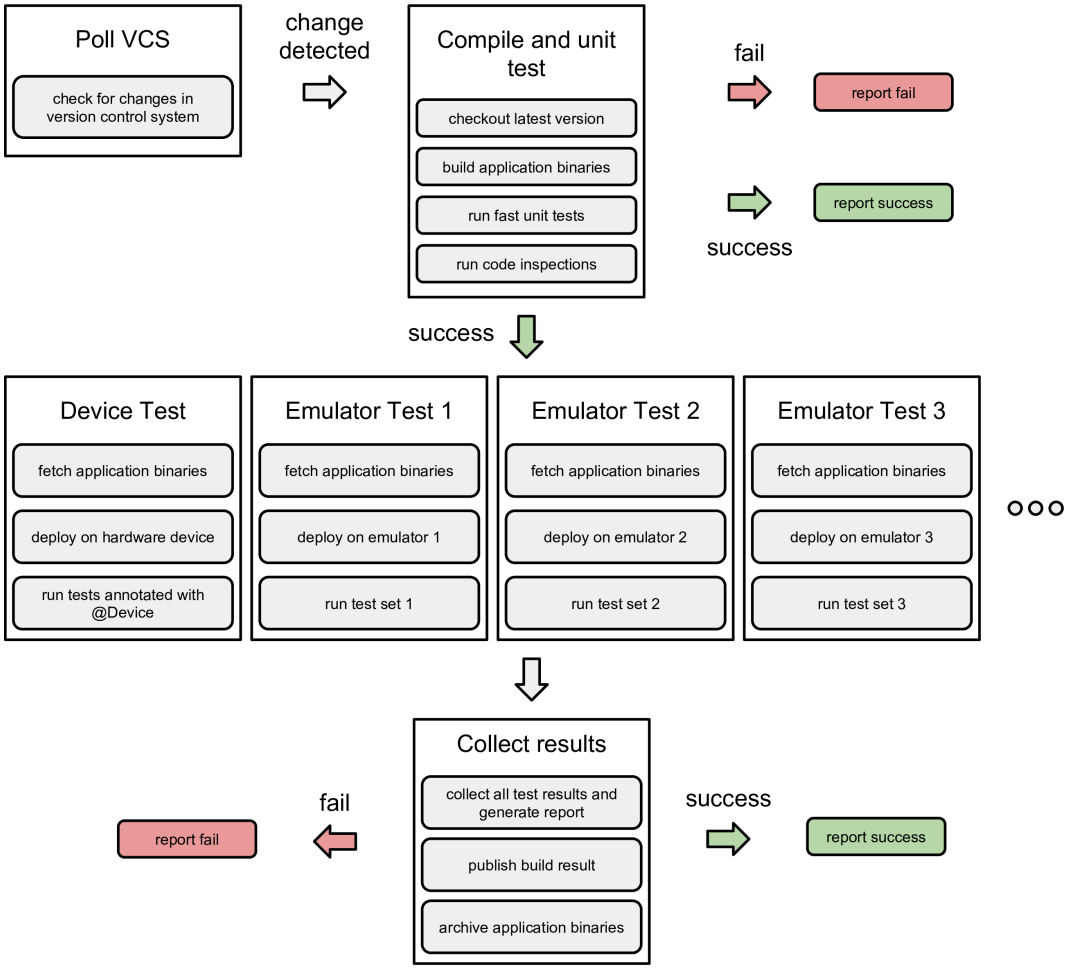

Figure 3.3: The Pocket Code CI build process

executed on a hardware device that is attached to the build server. Android emulators are automatically set up and configured by the Jenkins build server to ensure that the test environment stays the same for every build. Since Pocket Code requires OpenGL for full functionality the emulators are hardware accelerated and support GPU emulation to provide best performance. To eliminate any performance issues during the execution of several Android emulator instances in parallel, a high performance machine is important. A slow build server provides a large overhead and is in fact one of the first issues a CI team should address.

3.3 The Catrobat CI process

The improvements that have been made to the CI environment as well as to the workflow itself have yielded a much more responsive and beneficial process that will be described in detail in this section.

All projects that have been added to the build automation process are periodically tested once changes are introduced into their mainline. This happens multiple times during the day. The prerequisite for any member that wants to introduce new code into the main version of their project is to issue a pull request on GitHub.

During the implementation of any new code developers trigger build jobs on the Jenkins build server on the branch they are working in. These jobs execute the entire test set of the individual project and report the result back to the developer by sending messages to the projects IRC channel. As soon as the developed changes are believed to be ready to be merged into the mainline a pull request will be issued by the developer. To make

it easy for core members to review the request, developers include a link to the latest successful build on their branch in the description of the pull request. GitHub provides an easy to use and intuitive UI for pull requests that shows all the differences between the mainline version and the version that is introduced to it via the pull request. Notifications are sent out as soon as a new pull request has been created.

A core team member reviews the changes and marks possible areas of improvement. GitHub allows commenting on the individual lines of changed code which makes it very easy for core members to point out these areas.

Creating a new pull request automatically merges the target branch with the changes it contains. This version is temporarily accessible via a new Git reference until the pull request is accepted. This way merge conflicts are pointed out at the time the pull request is created since GitHub will automatically mark the pull request as "Unmergeable pull request". In this case the conflicts have to be resolved before any further progress. To ensure that the introduced changes work with the latest mainline version, the creation of a new pull request automatically triggers a build job that executes a fast set of unit tests and code inspections. The automatically triggered test job checks out the before mentioned temporary version that GitHub creates and runs its assigned build steps and test sets against it. The result of this job determines the state of the pull request. If the job fails, core members can immediately see that the changes should not be merged. Should there be any problems with the introduced code, the responsible developer has to fix the detected problems and commit the changes again. A commit to the same branch the pull request was issued from will automatically update the pull request and therefore automatically trigger the Jenkins build job. As soon as the pull request is marked as being ready to merge a core team member accepts it, thereby automatically merging the changes to the mainline.

Since the Jenkins build server is constantly watching the mainline for any changes, the acceptance of a pull request triggers another build job that executes a large test set including unit test as well as functional UI tests. In Pocket Code's case this job is set up as a build pipeline that is composed of multiple build jobs that carry out different parts of the process. The base job is constantly checking for updates on the source code repository and starts its routine as soon as changes are pushed to the master branch. This job sets up the environment that all consecutive jobs work on by executing the following steps:

- **Check out the latest version of the master branch**
- **Check out the latest version of the build scripts and utility scripts**
  - These are located in a separate repository.
- **Compile the project under test and all test projects**
- **Build the Android application files for all the projects**
  - A separate executable for the project under test and every test project is built in this step.

The build and utility scripts include additional steps for preparing the test environment including starting a self implemented Bluetooth server or configuring several Android virtual devices and the corresponding Android debugging bridge connections.

If one of the above mentioned steps should fail, the execution of any further steps is canceled and the build is considered to have failed. All issues that may have occurred during these steps have to be fixed and committed to the respective repositories in order to restart the build pipeline. If all steps completed successfully, the base job will trigger the first sub job which executes the following steps:

- **Run a set of unit tests**
  - This is a self implemented set of tests that checks whether the source code complies with the project specific coding guidelines.
- **Run static code inspections using Android lint**[7]
- **Run coding standard checks using Checkstyle**[8]
- **Check source code for bugs using PMD**[9]
  - PMD analyzes source code and can point out potential programming errors.
- **Check for bugs using FindBugs**[10]
  - FindBugs works on bytecode and is therefore complementary to PMD.

These steps are also mandatory. Therefore the build will be marked as having failed if any of the above mentioned steps should fail.
If all steps succeeded to this point, the base job triggers the execution of the Android instrumentation test sets. In order to complete this step as fast as possible several sub jobs are triggered simultaneously. These sub jobs are distributed among multiple Jenkins slave nodes and are set up to test a specific part of the test sets. All test cases that require a real Android device are marked using Java annotations so they can be filtered and executed on a suitable slave node.

The results of all above mentioned steps are archived. The base job fetches all archived data from the jobs it triggered and publishes the results of all tests and code inspections on the web dashboard on the Jenkins build server. Additionally the overall build result is published via email and IRC. The

[7] `http://developer.android.com/tools/help/lint.html/`
[8] `http://checkstyle.sourceforge.net/`
[9] `http://pmd.sourceforge.net/`
[10] `http://findbugs.sourceforge.net/`

result history of unit tests and code inspections of each project is published in form of trend graphs as shown in figure 3.4. These graphs provide a quick and illustrative insight on the current health of the individual projects.

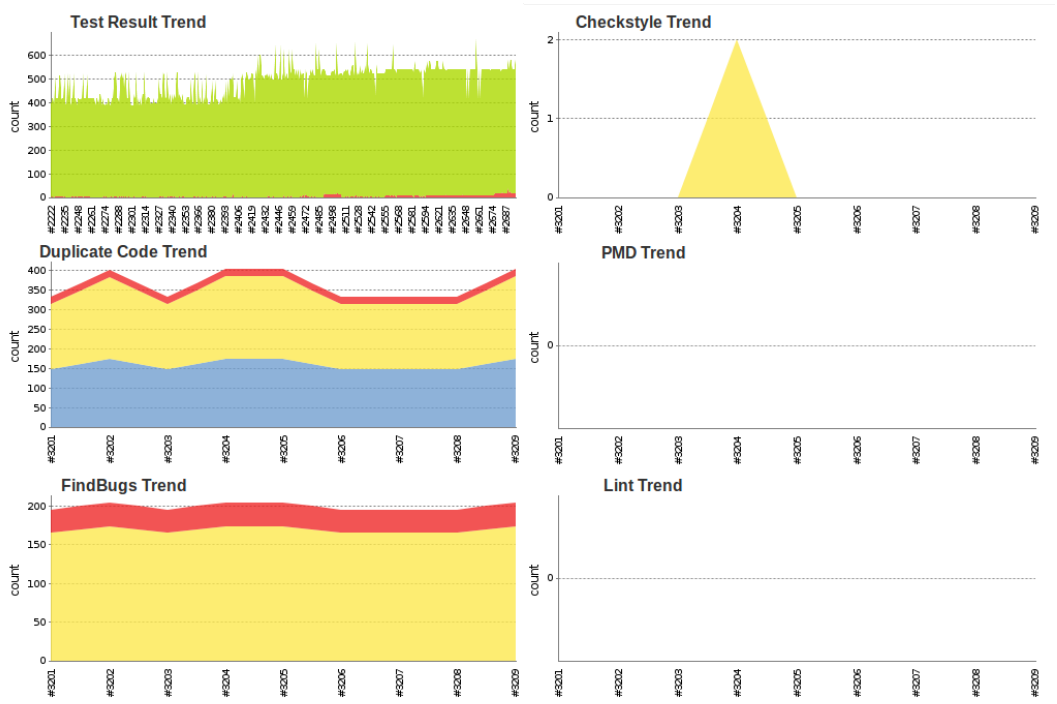

Figure 3.4: Trend graphs created by the Jenkins build server

3.3.1 Improvements

The introduction of the Jenkins master slave system has brought the capability of parallelization. This allows multiple different projects to execute their build jobs at the same time resulting in less time that is lost waiting idly in the execution queue.

The Pocket Code integration build job had an overall execution time of over 2 hours since every step was executed sequentially. The results of all test runs and source code inspections where only available after the job had finished. The separation of the build process into multiple jobs and the parallel execution has brought down the overall execution time to 40 minutes. Intermediate results about code inspections and fast unit tests are available after 10 minutes. Figure 3.5 illustrates the difference in the average build time of the former Pocket Code build job and the pipelined build job that has been introduced. The green bar marks the phase in which the switch between the two jobs was executed. Areas without any data points represent the periods in which the jobs were disabled and not yet active respectively. Spikes in the graphs are the result of configuration errors, network issues or other system failures.

The reduced execution time of the pipelined integration build job has an immediate impact on all other projects that use the Jenkins build server for their automated builds. Figure 3.6 shows the trend of the average build time measured across all build jobs of all these projects. Again, the green bar indicates the phase where the refactored build process has been introduced. A significant drop of the overall build time has been achieved.

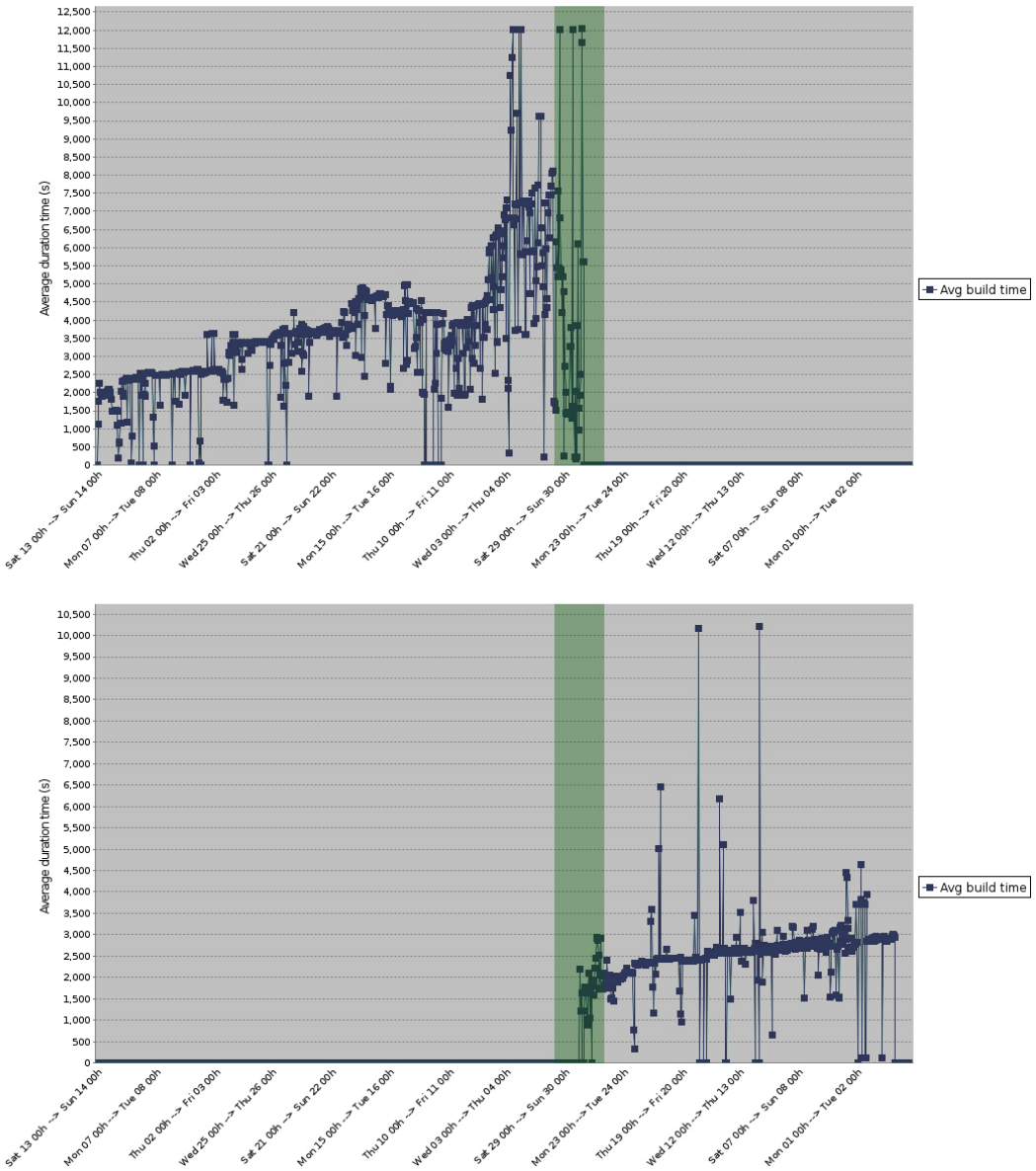

Figure 3.5: The average build time of the existing Pocket Code build job compared to the current pipelined build job

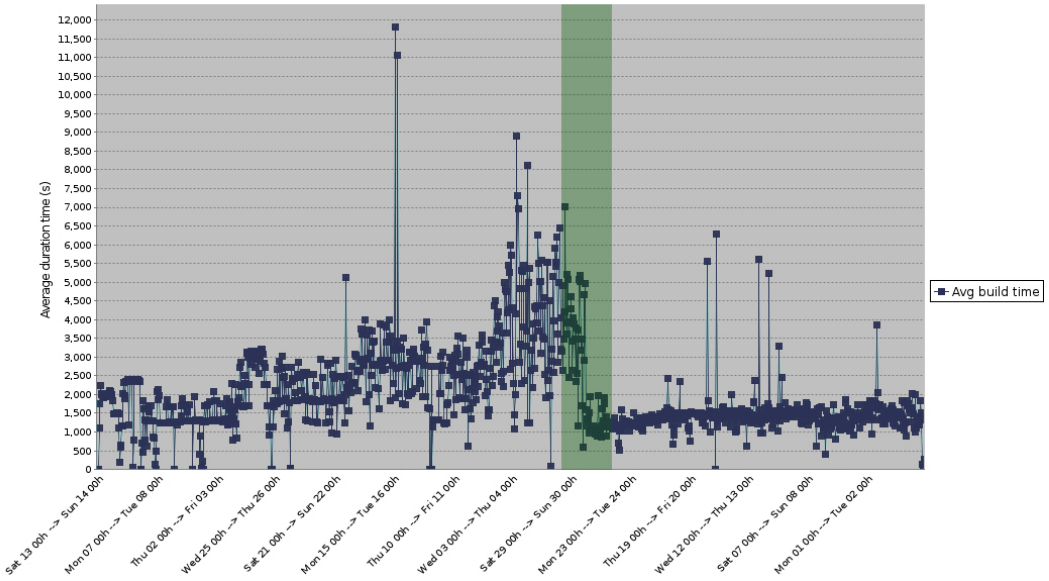

Figure 3.6: The average build time measured across all build jobs of all projects

4 Future work

Future efforts on further improving and optimizing the continuous integration process of the Catrobat project should be directed both at the process itself as well as the tools that are used. The following sections describe possible areas for improvement.

4.1 Keeping it simple

Continuous integration can not be done without a team that is willing to adopt the practices that define it. The project is guided by a very dynamic team meaning that developers come and go at all times. The process should therefore be kept as easy to adopt as possible to provide the least overhead possible. The trade-off between extending and overcomplicating the process is an important aspect to consider.

4.2 Build tools

Pocket Code started off using Ant to build the project. At the time this was the standard build process for Android projects. The build scripts used by Ant are written in XML which can lead to unnecessary large and complicated files that are hard to maintain by developers with little experience in working with Ant. The majority of the team members is working with IDEs like Eclipse that mask the build process well enough for the user to be able to build a project without having to know about the details of the build process.

With the growing popularity of the Android Studio IDE a build tool called Gradle is getting increased attention. Gradle aims at combining "the power and flexibility of Ant with the dependency management and conventions of Maven into a more effective way to build" [18].

Gradle build files are written using a Groovy based domain specific language (DSL). [18] A domain specific language makes it much easier for developers to write and maintain the build scripts.

4.3 Test suite separation

As mentioned before, the parallelization of build jobs requires the individual jobs to be independent. The Pocket Code integration build process distributes the entire set of UI tests on several Android devices simultaneously by splitting it into smaller sets. This separation is done via Java packages. While this already provides a major reduction of the test execution

time the distribution should be carried out more dynamically. Since the parallel build process can only be as fast as its slowest sub job, a package containing a large number of slow test cases would slow down the process. The execution time of the individual test sets should approximately be the same. Based on the previous execution times of the individual test cases these sets can be assembled by the build server at compile time. The maximum execution time of each set should be around 10 minutes.

4.4 UI testing

The Pocket Code project uses a large amount of UI tests during the integration process. These tests are written using the Robotium test automation framework. While this test suite provides good feedback about the functionality of the application as well as documentation of the code, it slows down the integration process significantly. In addition to the extension of the execution time of the automated process the test suite also requires a large amount of manual maintenance effort in case the application UI is changed. Moving UI elements to different views or changing the type of navigation elements requires a large number of test cases to be adapted to these changes.

Using UI crawling/ripping techniques as described in [2] and [3] could minimize the manual effort for UI testing the application while maintaining a structured approach. The major advantage of this approach over Monkey testing is that by exploring the UI the crawling algorithm generates a representation of it in form of a tree that can be used to automatically generate test cases. This would greatly reduce the maintenance overhead of the UI test suite and further automate the process.

4.5 Reduce test flakiness

Tests that randomly fail keep developers from searching for the reason of failure after some time. A failing build should never leave a team member thinking "It's OK, it will pass next time."

When using Robotium the result of an Android UI test is often influenced by timing issues. If a button has not yet finished drawing itself onto the screen the test will fail when it tries to perform a click on it. Robotium uses methods like *sleep(time)* or *waitForActivity()* as a workaround for these issues. However the time to wait for the UI to be ready very much depends on the performance of the device the test is running on. A device independent optimal value for *time* is therefore almost impossible to find. Additionally using *sleep(time)* can greatly extend the execution time of the test suite and therefore delay the feedback process.

Espresso [1] is a UI testing framework that provides an easy to use API on top of the standard Android instrumentation framework. Espresso eliminates sleeps by synchronizing background and UI threads and its introduction could reduce the execution time of the UI test sets. [31]

4.6 Testing on multiple devices

The functionality of Android applications can greatly differ between individual smartphone manufacturers and Android OS versions. Therefore it is important to test the application on multiple different device setups.

[1] `https://code.google.com/p/android-test-kit/wiki/Espresso`

Spoon [2] is a tool developed by Square Inc. [3] that can be used to automatically distribute Android tests among multiple devices simultaneously. [20] It generates a clear and easy to read HTML report that can include screenshots of the application during the test execution. Spoon makes it easy to quickly find device specific errors. However, in order to cover a large variety of different devices, each device has to be acquired and maintained locally.

Cloud based services like Testdroid[4] offer the possibility to access a wide variety of devices via the web. Not only does this approach solve the problem of acquiring hundreds of real mobile devices, it also enables developers to test from anywhere in the world without having to set up the necessary build environment.

[2] http://square.github.io/spoon/
[3] https://squareup.com/
[4] http://testdroid.com/

5 Conclusion

The growing complexity of the individual projects of Catrobat and the growing scale of the organization required a refactoring of the existing quality assurance process. The context of mobile application development presents many challenges to the theoretical concepts of software testing and continuous integration. Traditionally proposed test ratios that recommend keeping UI tests at a minimum due to their increased execution time and maintenance difficulty need to be reconsidered in order to accurately test mobile applications. This experience is shared with other projects as described in [24].

The automation of the testing process is key to effectively and constantly assure the quality of the software in its current state. Including all system and regression tests in this automated process is a very important aspect for maintaining a stable, well tested and release ready version of the project. [25] Using continuous integration, this automated process is carried out frequently and with every change that is made to the existing version. At the same time it is important to plan how to facilitate these automations in order to get the most benefit out of it. With the goal to make it possible to include all projects of Catrobat in the automated continuous integration process while keeping it fast without having to cut back on testing efforts came the realization that the existing integration workflow and the CI build environment had to be refactored. The work that has been done has

drastically reduced feedback delays for all projects that have already been included in the CI workflow. At the same time it has been ensured that the addition of further projects would not negatively impact this existing workflow.

As mentioned in [24], there are still limitations to what can be fully automated. Usability tests and localization testing should still be executed manually to a certain amount.

Catrobat's continuous integration process is still iteratively evolving into a core aspect of all its projects. It has become a critical part in ensuring the software quality. The work that has been done in providing a scalable environment enables future extensions to the process. The main goal of any future approach in extending the process should be to keep it as accessible and easy to adopt as possible. Since this process is driven by the whole development team, one of the main aspects for practicing continuous integration successfully is motivation. Developers must be able to see and experience the benefits of the process in order to be willing to carry out the required steps that define it. A scalable test environment that allows all projects to be included in the CI process was a large step towards further making CI a core part of Catrobat.

Bibliography

[1] Open Handset Alliance. *FAQ*. URL: http://www.openhandsetalliance.com/oha_faq.html (visited on 11/17/2014) (cit. on p. 25).

[2] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. "A GUI Crawling-Based Technique for Android Mobile Application Testing." In: *ICST Workshops*. IEEE Computer Society, 2011, pp. 252–261. URL: http://dblp.uni-trier.de/db/conf/icst/icstw2011.html#AmalfitanoFT11 (cit. on p. 53).

[3] Domenico Amalfitano et al. "A toolset for GUI testing of Android applications." In: *ICSM*. IEEE Computer Society, 2012, pp. 650–653. ISBN: 978-1-4673-2313-0. URL: http://dblp.uni-trier.de/db/conf/icsm/icsm2012.html#AmalfitanoFTCI12 (cit. on p. 53).

[4] Scott Chacon. *Pro Git*. 1st. Berkely, CA, USA: Apress, 2009. ISBN: 1430218339, 9781430218333 (cit. on p. 10).

[5] Paul Duvall. *Automation for the people: Continuous Integration anti-patterns*. 2007. URL: http://www.ibm.com/developerworks/java/library/j-ap11297/index.html (cit. on pp. 16–18).

[6] Paul Duvall, Stephen M. Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007. ISBN: 0321336380 (cit. on pp. 3, 7, 9, 40).

[7] Martin Fowler. *Continuous Integration*. 2006. URL: http://martinfowler.com/articles/continuousIntegration.html (cit. on pp. 7, 9, 11).

[8] Martin Fowler. *DeploymentPipeline*. 2013. URL: http://martinfowler.com/bliki/DeploymentPipeline.html (cit. on p. 40).

[9] Martin Fowler. *FeatureBranch*. 2009. URL: http://martinfowler.com/bliki/FeatureBranch.html (cit. on pp. 11–13).

[10] Google. *Activity*. URL: http://developer.android.com/reference/android/app/Activity.html (visited on 04/14/2014) (cit. on p. 29).

[11] Google. *Building and Running*. URL: http://developer.android.com/tools/building/index.html (visited on 04/14/2014) (cit. on pp. 27, 28).

[12] Google. *Creating a Fragment*. URL: http://developer.android.com/guide/components/fragments.html (visited on 10/28/2014) (cit. on pp. 28, 30).

[13] Google. *Using the Emulator*. URL: http://developer.android.com/tools/devices/emulator.html#limitations (visited on 11/13/2014) (cit. on p. 31).

[14] Jesper Holck and Niels Jørgensen. "Continuous Integration and Quality Assurance: a case study of two open source projects." In: *Australasian J. of Inf. Systems* 11.1 (2003). URL: http://dblp.uni-trier.de/db/journals/ajis/ajis11.html#HolckJ03 (cit. on p. 19).

[15] Jez Humble. *Continuous Delivery with Jez Humble*. 2012. URL: http://www.youtube.com/watch?v=IBghnXBz3_w&feature=youtu.be&t=11m42s (visited on 11/27/2014) (cit. on p. 11).

[16] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. 1st. Addison-Wesley Professional, 2010. ISBN: 0321601912, 9780321601919 (cit. on pp. 19, 40).

[17] GitHub Inc. *refactor versionName*. URL: https://github.com/Catrobat/Catroid/pull/1023 (visited on 08/04/2014) (cit. on p. 40).

[18] Gradleware Inc. *What is Gradle?* URL: http://www.gradle.org/ (visited on 04/14/2014) (cit. on p. 52).

[19] OpenSignal Inc. *Android Fragmentation Visualized*. 2013. URL: http://opensignal.com/reports/fragmentation-2013/ (visited on 04/14/2014) (cit. on p. 26).

[20] Square Inc. *Spoon*. URL: http://square.github.io/spoon/ (visited on 04/28/2014) (cit. on p. 55).

[21] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. "Real Challenges in Mobile App Development." In: *ESEM*. IEEE, 2013, pp. 15–24. ISBN: 978-0-7695-5056-5. URL: http://dblp.uni-trier.de/db/conf/esem/esem2013.html#JoorabchiMK13 (cit. on p. 32).

[22] B. Kirubakaran and V. Karthikeyani. "Mobile application testing - Challenges and solution approach through automation." In: *Pattern Recognition, Informatics and Mobile Engineering (PRIME), 2013 International Conference on*. Feb. 2013, pp. 79–84. DOI: 10.1109/ICPRIME.2013.6496451 (cit. on pp. 32, 33).

[23] Lasse Koskela. *Test Driven: Practical Tdd and Acceptance Tdd for Java Developers*. Greenwich, CT, USA: Manning Publications Co., 2007. ISBN: 9781932394856 (cit. on pp. 14, 15).

[24] Felix Krueger. "Field Report: Test Automation and Quality Assurance in the Context of Multi-Platform Mobile Development." In: *Testing Experience* 27 (2014), pp. 27–29 (cit. on pp. 56, 57).

[25] Prasad Ramanujam, Alisha Bakhthawar, and Mathangi Pollur Nott. "Demystifying DevOps Through a Tester's Perspective." In: *Testing Experience* 27 (2014), pp. 50–52 (cit. on p. 56).

[26] R. Owen Rogers. "Scaling Continuous Integration." In: *XP*. Ed. by Jutta Eckstein and Hubert Baumeister. Vol. 3092. Lecture Notes in Computer Science. Springer, 2004, pp. 68–76. ISBN: 3-540-22137-9. URL: http://dblp.uni-trier.de/db/conf/xpu/xp2004.html#Rogers04 (cit. on pp. 20–25).

[27] Tina Schweighofer and Marjan Heričko. "Mobile Device and Technology Characteristics' Impact on Mobile Application Testing." In: *SQAMIA 2013 Software Quality Analysis, Monitoring, Improvement, and Applications*. Sept. 2013, pp. 103–108 (cit. on pp. 32, 33).

[28] John Ferguson Smart. *Jenkins - The Definitive Guide: Continuos Integration for the Masses: also Covers Hudson.* O'Reilly, 2011, pp. I–XXII, 1–380. ISBN: 978-1-449-30535-2 (cit. on p. 14).

[29] Venkatesh Sriramulu et al. "Mobile Test Automation: Preparing the Right Mixture of Virtuality and Reality." In: *Testing Experience* 27 (2014), pp. 46–49 (cit. on pp. 31, 32).

[30] Mike Wolfson and Donn Felker. *Android Developer Tools Essentials: Android Studio to Zipalign*. O'Reilly Media, Inc., 2013. ISBN: 1449328210, 9781449328214 (cit. on p. 25).

[31] Valera Zakharov. *android-test-kit*. URL: https://code.google.com/p/android-test-kit/wiki/Espresso (visited on 04/28/2014) (cit. on p. 54).