Adapting and Implementing Continuous Integration Principles and Scrum:

On Automated Build and Deployment of a Complex Android Software System

sw609f15

Andreas Petersen, Mads Ellersgaard Kalør,
Michael Toft Jensen, Søren Bøtker Ranneries

AALBORG UNIVERSITY
STUDENT REPORT

Attributions

- The wrench icon on the cover is designed by Vecteezy.com user carterart
- The hard hat icon on the cover is designed by Vecteezy.com user nightwolfdezines
- The server icon on page 86 is designed by Chad Remsing
- The wireless router icon on page 86 is designed by Wilson Joseph
- The tablets icon on page 86 is designed by Pham Thi Dieu Linh

Department of Computer Science
Selma Lagerlöfs Vej 300
DK-9220 Aalborg Ø
http://www.cs.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Adapting and Implementing Continuous Integration Principles and Scrum: On Automated Build and Deployment of a Complex Android Software System

Theme:
Developing Complex Software Systems

Project period:
Spring semester 2015

Project group:
sw609f15

Participants:
Andreas Petersen
Mads Ellersgaard Kalør
Michael Toft Jensen
Søren Bøtker Ranneries

Supervisor:
Kim A. Jakobsen

Copies: 6

Pages: 120

Date of completion:
May 27, 2015

Abstract:

Continuous integration and deployment in a complex software system poses many challenges. We present a configuration of continuous integration and a development method for a large team of 6th semester Software students at Aalborg University working on the GIRAF app suite. The suite aims at easing the lives of people with Autism Spectrum Disorders and their caregivers. The project is organized according to Scrum of Scrums, comprising 15 groups of 1–4 students. The code base is inherited from previous semesters. The caretakers require a working product. We setup a tool, Jenkins, to automatically build and test apps and libraries that are developed for the project whenever changes are made to the code. The apps are automatically tested on physical tablets wirelessly and deployed to Google Play. Libraries are published to a Maven repository. Monkey tests are run on all GIRAF apps. We significantly improve the build environment compared to what we start with. The development method we specify together with the improved build environment enable groups to deliver what the stakeholders want.

Summary

Graphical Interface Resources for Autistic Folk (GIRAF) is a software suite developed by 6^{th} semester students from Software Engineering at Aalborg University. The development is a collaborative effort among groups of these students. The GIRAF software aims to assist citizens with Autism Spectrum Disorders (ASD) by easing their and their caregivers' everyday activities. GIRAF consists of Android apps meant for direct usage by the citizens with ASD, their caregivers, and some for administrative purposes.

One of the great challenges and learning goals of this project is to successfully work together on a large software system. Collaboration among groups is an important part, as this eases and accelerates the development. There are 15 groups with 1–4 students, working in *Scrum of Scrums*. The development on the software system is split into four sprints, each with a sprint planning and a sprint review meeting. At the sprint planning meeting, each group selects a fitting number of items from the product backlog, which they complete during the sprint. We, group sw609f15, are responsible for the overall software development method. We primarily work on continuous integration and the build environment supporting the software system.

In sprint 1, we set up the basic automated build in Jenkins. This includes running automated tests and generating documentation nightly. Additionally, we discuss various Git merging strategies related to continuous integration.

In sprint 2, we add a number of automated tests to compensate for missing unit tests in the individual GIRAF projects, including monkey testing. We make the continuous integration server create and publish test coverage reports for developers. We change the way internal dependencies are managed together with the Git-responsible group by replacing cumbersome Git submodules with a Maven repository.

In sprint 3, we evaluate and refine the development method based on the experiences we have had during the previous sprints. We do this in collaboration with another group. In addition, we make monkey tests use a local test database with dummy data to avoid downloading the full database.

In sprint 4, we further improve the test environment by replacing the emulator with physical devices, enabling us to test on a wide range of devices. This also decreases the overall time it takes to build a job in Jenkins.

In conclusion, we significantly improve the build environment — in particular the continuous integration platform. The developers now have a simple-to-use development environment, which aids in improving the stability of the software. In addition, we define and refine the development method followed by all groups in the development of GIRAF. The GIRAF project is in a significantly more stable state than before, and there is now a uniform experience across all apps.

Andreas Petersen

Mads Ellersgaard Kalør

Michael Toft Jensen

Søren Bøtker Ranneries

Preface

This report is a product of a bachelor project by four Software students at Aalborg University. The title of this semester project is Developing Complex Software Systems.

It is expected that the reader has a knowledge of computer science at the same level as a 6^{th} semester Software student at Aalborg University in order for the reader to benefit the most from reading this report.

Reading Guide

The report is comprised of several parts. There is a preliminary part which describes the framework for the project. There are four sprint parts, which chronologically details the work done during the project. The last part contains a conclusion and recommendations for future developers. Developers of next year will mostly be interested in the first and the last part.

Personal pronouns refer to the authors of this report. All figures in the report are made by the authors.

Citation Style

All references throughout the report are in IEEE style. Every source in lexicographical order is assigned the consecutive number in which it is referred to. The bibliography can be found at the end of this report on page 121.

Contents

Preliminaries **3**

1 Introduction 3

2 Software Development Method 5

3 Software Configuration Management Plan 15

Sprint 1 **19**

4 Sprint Planning 21

5 Setting Up Automated Build 25

6 Sprint Review 33

Sprint 2 **37**

7 Sprint Planning 39

8 Improving Automated Build 43

9 Improving Build Times 49

10 Sprint Review 63

Sprint 3 **65**

11 Sprint Planning 67

12 Development Method Improvements 69

13 Making Monkey Testing Work 73

14 Sprint Review 77

Sprint 4 **79**

15 Sprint Planning 81

16 Further Improving Build Times 85

17 Improving Monkey Testing 101

18 Servicing Developers 107

19 Sprint Review 111

Conclusion **115**

20 Project Evaluation 115

21 Recommendations for Future Developers 119

Bibliography 121

Appendix A Exact Build Times of Launcher App 125

Appendix B Gradle Plugins 133

Appendix C Dependency Workflow Guide 139

Appendix D Git Hooks 141

Appendix E Download and Install APKs Powershell Script 147

Appendix F Miscellaneous Server Scripts 151

Appendix G Continuous Integration Guideline 153

Appendix H UI Test Guide 155

Appendix I Jenkins Structure 157

Appendix J Wireless Router Setup 161

Appendix K Process Chapter from sw601f15 163

Preliminaries

1 Introduction

Every year, children and young adults are diagnosed with Autism Spectrum Disorders (ASD) [44], [45]. There are no known cures, and treatment focuses on improving independence and quality of life [43]. This project is a collaboration between the Software students on 6th semester at Aalborg University and several institutions, located in Aalborg Municipality, which offer care to children or adults who suffer from ASDs. The goal of the project is to ease the workload of the caregivers at these institutions and help persons with ASDs become more independent.

Today, caregivers have a selection of tools and methods which they use in the treatment of the ASDs. Some of these lend themselves to digitalization in the form of tablet applications (*apps*). The role of the caregivers in this collaboration is to select which tools and methods to digitize and to help us do it by providing knowledge of the domain. They will also be the primary users of the resulting system, so they act as the *external customers* in this collaboration. The role of the Software students in this collaboration is to understand their requirements and needs and develop the apps.

The overall system is called GIRAF (Graphical Interface Resources for Autistic Folk) and was initiated by Ulrik Nyman, Associate Professor at Aalborg University, in 2011. Since then, the 6th semester students every year have been working on the project. The main purpose of the system, which is nonprofit and open source, is assisting with planning and communication. Citizens with ASDs often have limited language skills and learn to enhance their communication with pictograms.

This year, 51 students work on the project, distributed into fifteen project groups: Eleven groups of four students, one group of three, one group of two, and two groups with a single member each. It is crucial that project groups work together towards creating a working system. This serves as a great challenge for us, as each group has previously only worked independently on their own projects. In addition, we take over an existing code base, which we have to understand and develop before we hand it over to the students next year. This requires that code is well documented and easy to understand. The tacit knowledge we acquire will not be passed to the next year's students unless we document it.

1.1 Bootstrapping the Project

While we do start the project with an existing code base, we need to define how we structure the work. We have an initial meeting where the apps are presented and the status of the project is explained by the semester coordinator.

This is followed by a discussion of how to organize the groups. The semester coordinator recommends an agile project with four iterations. We follow this recommendation as he has previous experience with the project and knows what has previously worked.

Because most groups have experience with the agile development method *Scrum*, and everyone has been introduced to it in the Software Engineering course, we initially decide to follow this method, which defines how work is prioritized and assigned. In addition, the semester coordinator explains that Scrum has worked well for the GIRAF project the previous years. The specific development method has evolved and been refined during the four iterations and Chapter 2 describes the method as it is by the end of the last iteration.

In the initial meeting we also assign groups with roles. These are described in more detail in Chapter 2. We undertake the responsibility for Jenkins [35], which is the tool used for continuous integration.

1.2 Project Goals

In the initial meeting with one of the external customers, it is identified that the most important requirement for the GIRAF project is that the existing apps become complete. The external customer sees no need for new apps, but instead that the bugs get fixed in the current apps so that they become stable. In addition, the customer wants the apps to have consistent user interfaces. It is the overall goal of the project to accommodate these desires from the customer.

Our goal is to improve build automation and the testing facilities of the project. This can help discover bugs and stability issues, which is important regarding the overall goal of the project.

1.3 Report Organization

In Chapter 2, we describe the software development method used across the multi-project as well as in our group. The method described is the final method which has been refined through each sprint. This is followed by Chapter 3, in which we present the Software Configuration Management Plan used in the multi-project. Elaborating on this, we describe the organizational context and the items and tools under configuration management.

The next parts of the report describe the four sprints. This includes descriptions of the user stories we solve and the technical work we make. In Sprint 1, we describe the reasoning behind the initial development method and the setup of continuous integration. In Sprint 2, we improve the build automation further by refactoring how internal libraries are referenced in projects. In the next sprint, Sprint 3, we analyze and evaluate the development method. In Sprint 4, we improve the test environment by testing on physical devices instead of emulators and prepare the handover of the project to next year's students.

Chapter 20 evaluates what we have experienced during this project and finally, in Chapter 21, we present a number of recommendations to future developers of the GIRAF project.

2 Software Development Method

One of the great challenges this semester is to coordinate the development process across all 15 groups. This chapter defines the development method as it looks at the end of the project.

Chapter Organization This chapter is organized in the following fashion:

- In Section 2.1 we describe how the multi-project is organized and introduce the hierarchy of the project;
- in Section 2.2 we describe the software development method in our group;
- in Section 2.3 we detail the responsibilities delegated to specific groups.

Chapter Abbreviations This chapter introduces the following abbreviations:

GUI	Graphical User Interface	**B&D**	Build & Deployment
DB	Database	**PO**	Product Owner

2.1 Multi-Project Organization Method

The multi-project consists of 15 groups. All groups collaborate on building the Giraf applications. The groups are organized into three subprojects: *Graphical user interface* (GUI), *databases* (DB), and *build and deployment* (B&D) by recommendation of the semester coordinator.

The GUI groups deal with bugs and implement new features in the front-end apps. The DB groups manage and maintain the database. The B&D groups control the tools supporting the building and testing environment as well as the deployment of the apps.

The multi-project groups work with an iterative development method, and the semester is divided into four iterations. No groups have any prior experience with the code base at the start of the multi-project, nor with the requirements of the external customers. Therefore there are many uncertainties associated with the multi-project. This makes the application of an agile method suited for the multi-project. Suppose the multi-project is organized in a traditional method. Then the multi-project groups will spend much time initially analyzing the current code base and future requirements, rather than actually work on making the code base work (a prime wish of the external customer). These requirements may change over time, which makes it difficult to deliver what the users want.

The agile method used for the multi-project is Scrum, and the groups are organized as Scrum of Scrums. Scrum of Scrums divides the developers into several layers of Scrum

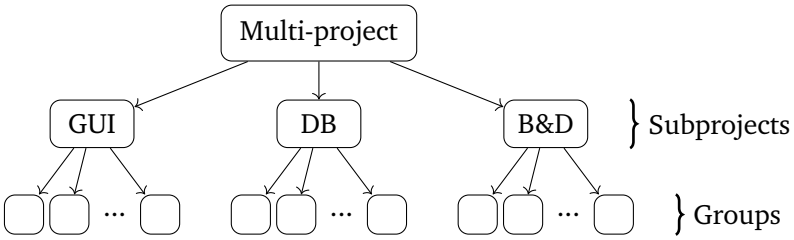


Figure 2.1: Illustration of multi-project organization

teams [41]. Figure 2.1 shows the structure of the multi-project organization, which consists of three levels:

Multi-Project Level The purpose of the overall level is to ensure that the roles, work products, and meetings are followed as intended. That is, that the development method is followed. A weekly meeting is held where the status of each subproject is discussed, and the development method and roles are evaluated. Every group has at least one representative at the weekly multi-project meetings.

Subproject Level Each subproject has the responsibility of solving a number of backlog items related to either GUI, DB, or B&D. Representatives from each group in a subproject have sprint planning and sprint review meetings with the other members of the subproject and hold at minimum two Scrum meetings a week. We choose not to meet every day because we think that it will steal too much time, compared to the benefits. It is common for such a meeting to be held two or three times a week [41].

Group Level Each group solves some of the backlog items of its subproject. While Scrum of Scrums dictates that Scrum is followed at all levels, this is not the case for this multi-project organization. This is of course a major deviation from Scrum of Scrums. When initially deciding the software development method, some groups expressed an interest in not running Scrum. Thus, each group is given the freedom to organize how they see fit. However, all groups are expected to fill in the work products and attend the meetings described later.

2.1.1 Work Products

Two work products from Scrum are used, and every group is expected to help updating and maintaining these:

Product Backlog The product backlog contains the items of value to the customers, typically features, but also other items [39]. On the multi-project level, there is a product backlog containing all items for the entire multi-project. The purpose of the multi-project level product backlog is to maintain the product backlog for the entire multi-project. The items in the product backlog are categorized according to

each subproject. This enables each subproject to manage their items themselves, removing unnecessary management from the multi-project level.

Release Backlog The release backlog contains the items that must be finished by the end of the current sprint [39]. These items are selected from the product backlog during sprint planning meetings that each subproject hold.

Since it is not required for each group to use Scrum at the group level, there is no common sprint backlog.

The product backlog and release backlog contain the following items:

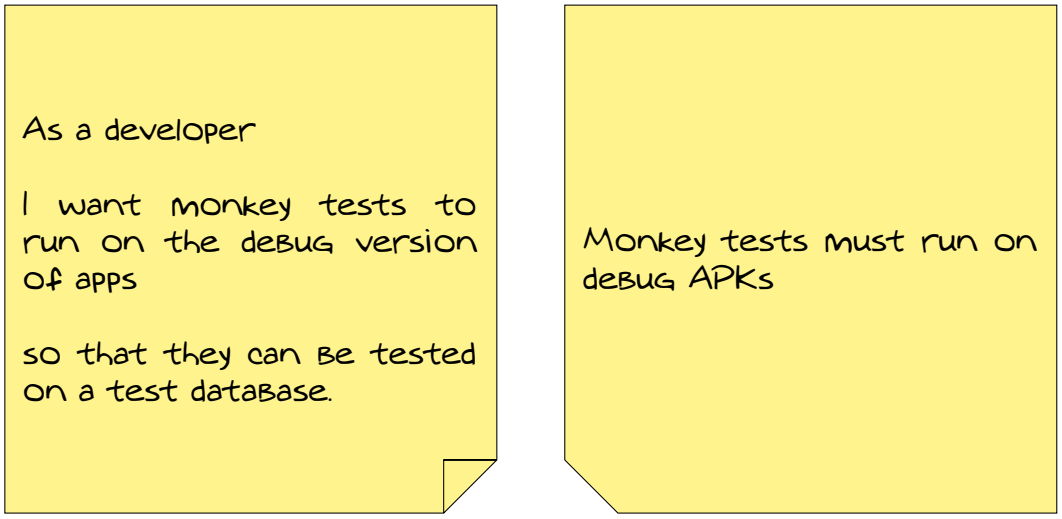
Feature Functional requirements (features) of the product are represented as user stories, as they are lightweight and fit well into the agile principles [55].

Bug Because bugs also represents behavior the user wants to be changed, they are considered no different from features [42], and so are represented as user stories.

Constraint A way of handling non-functional requirements, while not forcing it into a user story, is to turn the non-functional requirements into constraints [16, ch.16]. Most often, these constraints are related to e.g. performance, usability, and security. Constraints can either be written on separate constraint cards [16, ch.16], or written in the corner of a user story card if it is relevant only to that single user story [16, ch.7]. Due to the structure of multi-project, it makes sense for us to introduce two-level constraints. Multi-project constraints are relevant for the entire project as a whole, for example *the software is written in Java* or *all apps must be easy to internationalize*. Some constraints only apply to some areas of the multi-project, and these are handled by the individual groups. Examples of these are *the initial database sync must not take more than 5 minutes* or *the user manager must only sync data of the user logged in*. These constraints are written on either a constraint card or on the relevant backlog item.

Knowledge Acquisition Since no one in the multi-project has worked on it before, and because the multi-project is complex, it is sometimes needed to investigate things to know it if is worth investing time in. Knowledge acquisition items can be formulated as user stories, but do not need to [55]. Knowledge acquisition items are handled the same way as user stories. Estimation can be hard when investigating an unknown subject. The item can be investigated just enough to do further estimation by *timeboxing* an initial investigation. This practice is in XP called a *spike* [16].

Technical Work Sometimes effort has to be spent on something the users are not directly interested in, e.g. updating software on the developer machines, updating the database software on a server, or refactoring parts of the code. These items can be part of the product backlog [16]. They can also be formulated the same way as user stories [55] and are handled the same way.



(a) Front with user story **(b)** Back with conditions of satisfaction

Figure 2.2: Example user story

To summarize, the backlog items *features* and *bugs* are represented as user stories, whereas *knowledge acquisition* and *technical work* can be represented as user stories, but do not need to.

The following template, presented by Cohn [15], is used for user stories in the multi-project:

As a ⟨*type of user*⟩, I want ⟨*some goal*⟩ so that ⟨*some reason*⟩.

If the user story is suggested by developer, the source of the user story should be noted in case there are questions about it.

A user story must be sufficiently small for it to be completed in one sprint [15]. A user story must also have *conditions of satisfaction*. These are high-level acceptance tests for the user story and must be true for the user story to be completed. A user story can have multiple conditions of satisfaction [15]. Backlog items are prioritized by the product owner (possibly with help from the customers) [39]. An example of a user story can be seen in Figure 2.2.

2.1.2 Roles

Two roles from Scrum are used in the multi-project:

Scrum Master On the multi-project level, we are responsible for the development method. This means that we act as Scrum masters on the multi-project level. It is our role as Scrum masters to know and reinforce the sprint project goals and visions, and to ensure that the Scrum practices and values are followed on the multi-project level [39]. On the subproject level, one person is in charge of the

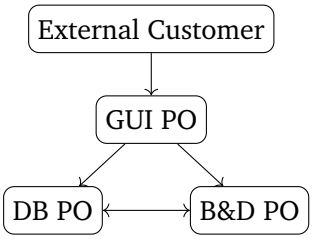


Figure 2.3: Illustration of product owners

meetings — they make sure that the Scrum meetings on this level are held. On the group level, each group is free to do what they see fit.

Product Owner The product owner (PO) is responsible for maintaining contact with the customers and being their representatives. It can be difficult to have a single PO with an external customer as customer in the multi-project. Therefore, the B&D and DB groups often do not have the external customer as their direct customer. Rather, these groups help the GUI groups achieve their goal by building solutions for them. The external customer is often not interested in the implementation details of build and database interfaces.

There are three POs in the multi-project, one for each subproject. The customer for the GUI PO is the external customer, whereas the customers for the DB and B&D POs are the groups of the other subprojects, as illustrated in Figure 2.3.

2.1.3 Meetings

Each week there is a meeting at the multi-project level. At this meeting the status of all groups in the multi-project is presented and the development method is evaluated. This ensures that all groups know the current status of the multi-project. This meeting differs from a regular Scrum meeting in that it includes a point on method evaluation, so that the development method can be continually improved.

Each subproject holds internal Scrum meetings as well. These subproject Scrum meetings are held at least twice a week. The subprojects organize these meetings themselves and use them to discuss things that are relevant only for the members of the subproject.

Since there are three product owners in the multi-project, three sprint review meetings and three sprint planning meetings are held between sprints. These meetings were proposed by us, following the recommendation from Bird and Davies [13], and afterward approved by the other groups.

Sprint Planning Meetings The purpose of the sprint planning meetings is to ensure that the product backlog is up to date and to select user stories for the release backlog.

GUI At this sprint planning only one representative from each GUI group participate as *pigs* (people who are allowed to talk), as well as the GUI PO and one

representative from the other PO groups (although they have a less important role at this meeting). All other people in the multi-project are *chickens* (people who are not allowed to talk). This meeting is held with the external customers. By making it so that primarily the GUI groups are pigs, the external customers are abstracted from internal development such as continuous integration and database management.

DB and B&D There are separate sprint planning meetings for DB and B&D, where each subproject establishes the needs of the other subprojects. These meetings are internal, and the external customers do not participate, since DB and B&D do not have these as direct customers. DB and B&D will primarily have the GUI groups as their customers, who need various services from the DB and B&D subprojects, in order for them to fulfill their own backlog items directly related to the external customers. However, it might be that DB needs something from B&D and vice versa. Again, this abstracts the external customers from internal development.

Sprint Review Meetings At the sprint review meetings, the work done in the sprint is presented to the customers.

GUI At this meeting the external customers are presented with the work of the GUI groups. Only the GUI groups participate as pigs at this meeting.

DB and B&D DB and B&D each hold a separate sprint review meeting where they show the groups in the other subprojects what they have done in a sprint. The external customers are not present at these meetings, so that they are not bored with internal details. The GUI sprint planning meeting is separated from the DB and B&D sprint planning meeting by a few days, such that the product owners have time to organize, and make the requirements propagate from GUI to the other subprojects.

Sprint Retrospective Meeting The purpose of this meeting is to learn from the past to improve productivity in the future; a time for diagnosis of the multi-project [28]. This meeting is held the Tuesday following a Sprint Review Meeting. In this meeting the previous sprint is discussed. There is extra focus on any problems regarding the development method during this meeting. Any concerns that are raised during this meeting are discussed, and possible solutions are suggested.

The sprint timeline is shown in Figure 2.4. Each sprint has approximately 8 full days per man. The exact sprint meeting dates are as follows:

- Sprint 1
  - GUI, DB, and B&D sprint planning meetings: February 15
  - GUI, DB, and B&D sprint review meetings: March 10
- Sprint 2
  - Sprint 2 GUI, DB, and B&D planning meetings: March 12

  - Sprint 2 GUI, DB, and B&D review meetings: April 8

- Sprint 3
  - Sprint 3 GUI planning meeting: April 8
  - Sprint 3 DB and B&D planning meetings: April 15
  - Sprint 3 GUI, DB, and B&D review meetings: April 27

- Sprint 4
  - Sprint 4 GUI planning meeting: April 27
  - Sprint 4 DB and B&D planning meetings: May 4
  - Sprint 4 GUI, DB, and B&D review meetings: May 20

Notice that the GUI sprint planning meetings in the first two sprints are not separated from the DB and B&D planning meetings, as this delay was introduced after sprint 2.

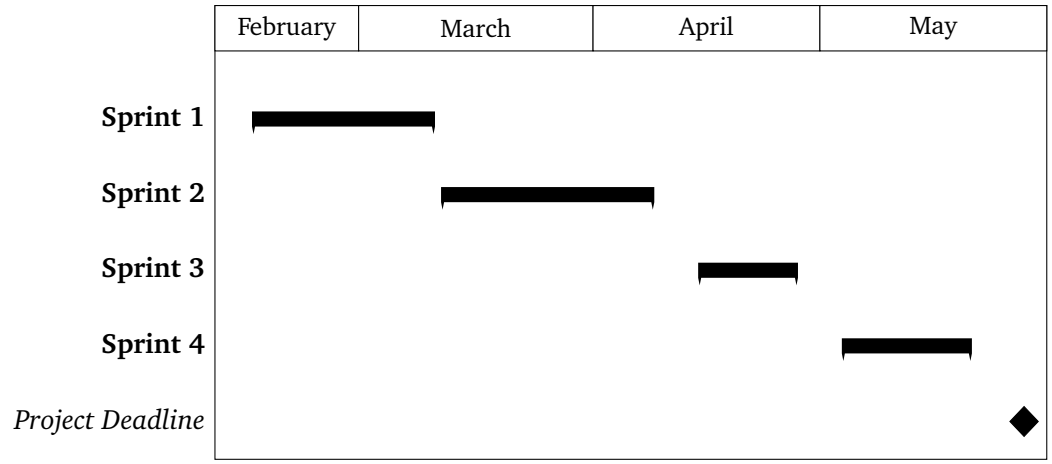


Figure 2.4: Gantt chart illustrating the sprint timeline

2.1.4 Office Hours

A vital part of working agile involves being able to easily communicate with all members of the multi-project. To ensure this all groups must be reachable on e-mail all weekdays 09:00–15:00, and preferably be in the group rooms. An exception for this is when there are courses.

2.1.5 Tools

To support the development method of the multi-project, an online project management tool, Redmine [53], is used. This tool was in use the previous year, and so this year the multi-project continues to use it. The main features are:

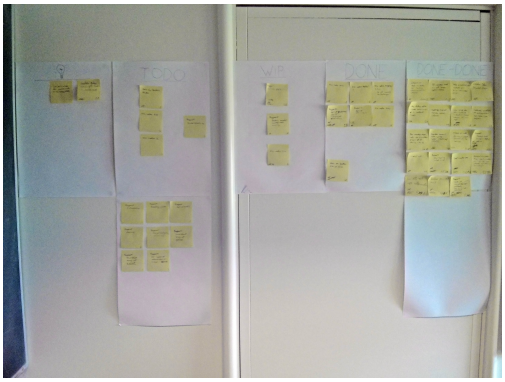


Figure 2.5: Our sprint backlog

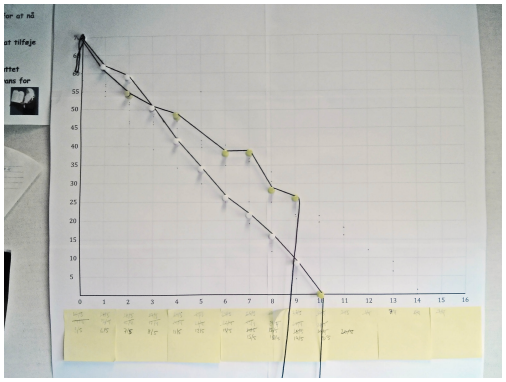


Figure 2.6: Our burndown chart

Wiki A wiki is used for various information, such as guides, how the development method is structured, meeting minutes, etc. Previous years had several separate wikis for each subproject. This year there is a single wiki providing central information.

Forum The forum is used for non-crucial communication. It is mostly used for non-work related discussions such as social events. Groups are encouraged to communicate to each other directly by going to each other's group rooms, rather than using the forum.

In addition to Redmine, Google Docs is used by the POs to maintain the backlog.

2.2 Software Development Method in Our Group

In our group we use Scrum as the development method, because it fits well into the multi-project development method. We have a Daily Scrum each morning where we summarize our work since last time and what we intend to do until the next Daily Scrum.

In addition to the previously described work products, we use a sprint backlog. After a sprint planning where backlog items have been selected by each group, we then split the backlog items selected by us into tasks and estimate those tasks using planning poker. Our unit of estimation is in half days, with the smallest unit being 1. This means that tasks that can actually be completed in less than a half day, may become overestimated. However, since there is an inherent imprecision in our estimations, this is not an issue. Our sprint backlog is physically placed on a wall, so that we can easily manage our tasks, seen in Figure 2.5. In addition to the sprint backlog we use a burndown chart to measure our progress, seen in Figure 2.6.

2.3 Group Roles in Multi-Project

The multi-project contains responsibilities selected by groups within the multi-project. These roles include management of Redmine, server, Git, and Jenkins:

Redmine General Maintaining users, roles, and plugins on Redmine as well as managing work across the other Redmine groups below.

Redmine Wiki Structuring and maintaining the wiki. They also read the content and make sure it is adequate, and notify the relevant groups if problems are found.

Redmine Forum Structuring and moderating the forum.

Redmine Task Tracking Structuring and maintaining the task tracker as well as helping people creating the items in the tracker. They also maintain guidelines for task creation.

Server Maintaining the server software and controlling access to the server. They install and upgrade software upon request and help solving issues requiring root access.

Git Maintaining the Git repositories and controlling access to these. Furthermore, they issue guidelines for Git usage.

Jenkins Maintaining the Jenkins setup. This is the responsibility of our group, and as such will be detailed in this report.

Process Supervising, developing, and refining the process method. This is also the responsibility of our group, and will as such also be detailed in this report.

Code Style Developing and encouraging a code style across all code.

Customer Relations Maintains customer relations. They review and send any email sent to the external customers.

Web Administrator Maintains the public website `http://giraf.cs.aau.dk`.

Android Guru Persons with experience developing Android apps, and who can be asked for help.

Google Analytics and Google Play Maintaining the Google Play listing of the software and generating guidelines for how to get crash reports from Google Analystics.

Graphics Developing a graphical style and enforcing this as well as creating the graphics upon request.

3 Software Configuration Management Plan

Software configuration management is an important part of any large software project. Configuration management can be defined as:

> *... the discipline of identifying the configuration of a system at distinct points in time for the purpose of systematically controlling changes to the configuration and maintaining the integrity and traceability of the configuration throughout the system life cycle* [58, ch.6, p.6-1]

The large size of the multi-project requires some form of software configuration management to handle and track the changes in the software. It can ease the workflow of developers and assure the quality of the product [58, ch.6].

Our role as a group is to manage the development method, and therefore we have the responsibility of creating a Software Configuration Management Plan (SCMP). An SCMP specifies how items are controlled, by who, and by what tools [58, ch.6].

This chapter describes how we plan and execute configuration management. We aim to make development easy for the developers, and therefore aim to automate as much as possible.

Chapter Organization This chapter is organized in the following fashion:

- In Section 3.1 we describe the organizational context of the multi-project;
- in Section 3.2 we identify which items are under configuration management and describe how they are managed;
- in Section 3.3 we describe the continuous integration practices we use;
- in Section 3.4 we list the specific tools we use for configuration management.

3.1 Organizational Context

In order to plan the software configuration management it is important to understand the organizational structure [58, ch.6]. The multi-project has a quite unique structure. The development is performed and managed by students who work in small groups that each acts as an independent entity. While the product is being developed for the clients, the groups have other priorities than customer satisfaction. Since the work is being done for free, the motivation of the groups is to work on something interesting such that they can write a good report detailing the work and get a good grade. While we can

suggest a process with specific procedures, we have no authority over the other groups. This means that the preferences of the groups weigh heavily when we select which software configuration management procedures to follow. In general, the consensus of the multi-project groups is that the less formal communication and bureaucracy the better. This is reflected in the choice of Scrum as the basis for the project management, and will influence all of the decision regarding the software configuration management process planning and execution.

3.2 Configuration Items

A configuration item is a piece of software or a combination of hardware and software which is managed as a single entity [58, ch.6]. The identified items and how they are managed are:

Requirements The requirements are managed by the subproject product owners as described in Section 2.1.

Released Applications The currently deployed version of all applications is controlled by the group responsible for Google Play. The current version of internally built applications is controlled by us through automatic building. Changes made to an application trigger building and testing. We ensure that the tests are run automatically, but it is the responsibility of the developers of each application to write the tests. The iOS applications developed during previous years are not under configuration management, as it was decided at the initial meeting that those applications will not be worked on.

Released Libraries The binaries of all libraries are stored in a Maven repository. When a library builds successfully the binary is uploaded to the repository as a snapshot version. When the developers of the libraries decide that a new release should be made available, new versions of the libraries are uploaded. This way several versions of library binaries are stored.

The Supporting Tools Gradle and Android Studio The developers on the multi-project use Android Studio as the main development tool for the Android applications. Android Studio uses Gradle as its build automation system. The versions of Android Studio and Gradle are maintained by the B&D groups and any decision to upgrade is theirs. When the multi-project started Android Studio and Gradle was upgraded. The used version of Android Studio is 1.0.2 and 2.2.1 for Gradle.

Application and Library Source Code The source code for the applications and libraries is controlled with Git [26]. Each application and library has its own repository. The external dependencies are managed with Gradle which gets the correct version of each external library from the Maven repository.

Source Code Documentation Documentation of source code is written by the developers of the software in Javadoc style. There is no formal process ensuring that it is

actually written, and so it is solely up to the developers to do so. Documentation is automatically generated.

3.3 Continuous Integration

Continuous integration is used to tie together the items under configuration management. Applications and libraries are automatically tested and built so we always have the most recent versions available for the customer. Developers integrate with the master branch frequently to avoid large, error-prone merge conflicts at the end of a sprint. Documentation is automatically generated daily so it is always up-to-date. Quality metrics of the source code, such as test code coverage and potential problems found using static code analysis, are automatically generated. To make continuous integration work in practice, developers are expected to integrate with the master branch daily, write automatic tests, and test code locally before integrating.

3.4 Tools

We use the following tools to manage the configuration items identified:

Git Git is used as the version control as mentioned previously. This was set up by previous years, and we continue to use their setup. The Git group manages Git.

Jenkins Jenkins is used to automatically build and test applications. Jenkins is detailed in Section 5.2. Jenkins is used for all applications throughout the multi-project, except for the iOS applications. We manage Jenkins.

Artifactory The Artifactory tool is a Maven repository that stores various versions of binaries of the libraries in the multi-project. Artifactory, managed by us, is described in more detail in Section 9.3.

Doxygen We use Doxygen to generate source code documentation. The use of Doxygen is described further in Section 5.4. Doxygen is managed by us, but the documentation itself is managed by the developers.

Sprint 1

4 Sprint Planning

The focus of this sprint is to bring the code base into a working state, as pointed out by the external customers. The first sprint planning differs from the remaining, because the development method at this point is not as well defined as later. In this sprint planning meetings are held on three levels: Multi-project, subproject, and group.

Chapter Organization This chapter is organized in the following fashion:

- In Section 4.1 we describe how we aggregate all known user stories from previous years;
- in Section 4.2 we describe the multi-project planning meeting, especially how user stories are selected;
- in Section 4.3 we describe the delegation of user stories at the B&D sprint planning meeting;
- in Section 4.4 we describe our own group sprint planning;
- in Section 4.5 we evaluate problems that arose during sprint planning.

4.1 User Stories Aggregation

The students from last year had no centralized product backlog, despite the fact that the previous year used Scrum as well. We therefore ask all groups at the start of the multi-project to read a report from the previous year and look for candidate user stories. These are then aggregated in a shared document. The user stories are categorized according to the subproject under which they relate to the most.

4.2 Multi-Project Planning Meeting

With all the reports of last year read and the backlog updated to reflect the status of the multi-project, the sprint planning can commence. All groups are present at this meeting, and some *pre-planning game* activities like project vision are discussed. The newly created backlog is reviewed and each subproject select the user stories they will work on and adds them to the release backlog for sprint 1. The stories are selected according to importance for the customer, which can be other groups as well as the external customer. The priority of user stories at this time is unclear, since at this time there has been no review of the backlog with the external customers. The priority is therefore the same as it was last year, but the customers' requirements may have changed since then.

4.3 B&D Sprint Planning

After the multi-project sprint planning meeting, the B&D subproject holds a sprint planning meeting. At this meeting, the B&D groups select user stories for their sprint release backlog from the B&D subproject release backlog. This is done such that each group has influence on what they are working on and commit themselves to getting it done. It is hard to know in advance how much work each story contains without dividing it into tasks. We choose not to divide the stories at this meeting, to keep it as short as possible at the cost of precise time estimates. The groups select stories until they feel they have a sufficient work load. The uncertainty of the stories is problematic, as groups may find that they are overburdened or find themselves out of work too soon. We select a single user story, namely *Continuous Build & Integration*. We commit ourselves to this user story because we think our knowledge about the development method is applicable in order to create an automated build and deployment process that supports the overall development method. We are aware that it is a very vaguely defined story, but there are many uncertainties about the multi-project in this initial sprint which makes it difficult to make more precise user stories.

4.4 Group Sprint Planning

After the B&D sprint planning meeting, we meet in the group and do the sprint planning for our group. We split the user story which we chose in the B&D sprint planning meeting and divide it into tasks. The tasks are added to the sprint backlog of our group. Then we estimate the tasks to see if we have a suitable workload for the sprint. In general a group will select more user stories if they estimate that they have too light a workload, or remove some user stories from the release backlog of the group if they are overburdened. In this case we have a large and vague story, so instead we adjusted the tasks included in the story to fit the available time.

The tasks are listed in Table 4.1, however the task estimation values are lost and are thus not available in the table. All tasks in the table are related to the user story *Continuous Build & Integration*. Tasks with a plus (+) are tasks that have been added during the sprint as they were discovered. Missed tasks are tasked that we did not manage to fulfill in this sprint. Rejected tasks are tasks that we have rejected during the sprint.

4.5 Response to Sprint Planning

When the first sprint planning was made, it was decided that each subproject would choose what to work on, by selecting user stories to put into the release backlog. The developers of last year documented the requirements of the apps in a traditional requirements document. This led to some confusion amongst the groups as it was not clear how these requirements should be converted to user stories for the release backlog and how these stories should be split into tasks for the sprint backlog. This uncertainty is addressed at

Task
Upgrade Jenkins
Upgrade Jenkins plugins
Automated build and test
Setup of roles in Jenkins
Setup of Javadoc in Jenkins
Investigate lint checking
Setup automatic lint checker in Jenkins
Investigate UI testing
Investigate monkey testing
Investigate Android unit testing
Setup Android unit testing
Setup Android unit testing in Jenkins
Investigate concolic testing
Enable Jenkins to start an Android emulator
Jenkins cannot start (+)
Write wiki entry on process
Write wiki entry on meetings
Write wiki entry on backlog
Write wiki entry on roles
Setup sending email to people breaking builds
Setup Android publisher plugin for Jenkins
Missed tasks
Setup monkey testing in Jenkins
Rejected tasks
Setup concolic testing in Jenkins
Write wiki entry about concolic testing

Table 4.1: Sprint backlog for sprint 1. The tasks are listed in no particular order.

the following multi-project meeting, where the terms are discussed. The product backlog and release backlog are inserted into Redmine by each subproject. In the B&D subproject this task is carried out by our group in collaboration with Group 5 (B&D product owner).

5 Setting Up Automated Build

Starting the first sprint we need to understand the systems given to us by previous semesters. We mainly look at setting up automated build, but also do some setup of the Redmine tool.

Chapter Organization This chapter is organized in the following fashion:

- In Section 5.1 we describe how we customize Redmine to suit our needs;
- in Section 5.2 we describe the platform for continuous integration, Jenkins, that we use in the multi-project;
- in Section 5.3 we explain how we set up automation of the builds and discuss version control branching strategies as well as automated testing and lint checking;
- in Section 5.4 we explain how we set up automated documentation generation.

5.1 Configuring Redmine

The developers of last year used Redmine for various project management functions, and we need to evaluate which of the functions to keep. One of these functions is an issue tracker which we choose to keep, but not as a traditional issue tracker. We customize it to contain the product backlog and release backlog. Redmine is not an ideal tool for this purpose, as the workflow of adding and updating tasks is slow and tedious. However it is what the multi-project has available without additional effort. We collaborate with the Redmine and issue tracker groups. We specify how user stories should be handled in Redmine, and the aforementioned groups implement the changes.

Also, a Gantt chart feature had been installed by previous years. The Scrum method does not advocate Gantt charts or other dependency charts [39]. As such, we collaborated with the Redmine group to remove this from Redmine. The other unused features have not been removed, since it was deemed unnecessary to spend time on doing so.

Redmine also contains other features such as a roadmap, burndown charts, and news. While these features are present on the main page, they are not used. The burndown chart in particularly is not used since no common sprint backlog is used, and as such the burndown chart cannot be used at the multi-project level.

We initially used Redmine as Scrum board within our group, but due to Redmine being very slow and tedious to update, we replace it with a physical Scrum board.

5.2 Jenkins: Continuous Integration Platform

An open source tool for continuous integration, Jenkins [35], was used by previous years. We will continue to use this as our continuous integration platform. It supports source control management tools such as Git [26], as well as build automation tools such as Maven [8]. It is extensible via numerous available plugins. Jenkins allows for a sophisticated continuous integration setup, however the setup by previous years is rather basic and we want to improve it in several ways.

5.2.1 Upgrading Jenkins and Plugins

We inherit the old installation which has not been updated in a long time. Jenkins itself and all the Jenkins plugins have updates available. We update everything to the newest versions.

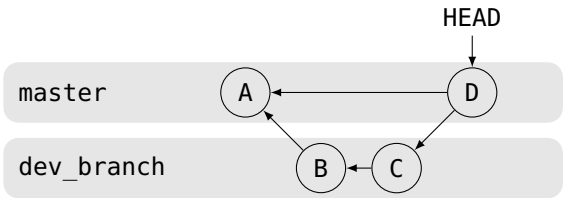
5.2.2 Setting up Roles in Jenkins

The inherited Jenkins installation is open to anybody. We do not find this sensible as we need to control the build process. Allowing everybody access will likely end in someone modifying a setting without our knowledge. Because we set up a mechanism for automatic build, other people do not need the option to start builds manually. It might even interfere with the automatic build, if other people have access to the Jenkins configuration.

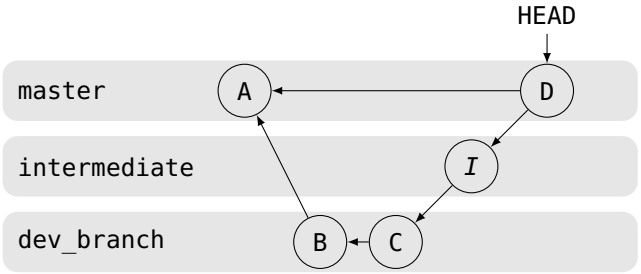
It is important that everyone can see the build process, however. According to Martin Fowler, it is important that everyone is able to see the state of builds and which parts of the overall system that are currently worked on [20]. In addition to this, we find it important that developers can follow the testing process of their new code, and how stable different the code is. This way, we are able to transfer human resources between code bases if needed, and it may work as a motivation for the developers to create stable builds. Because of this, we give developers read-only access to Jenkins, while we are the only people with write access.

5.3 Build and Test Automation

The inherited build project has no build automation. As a part of the automated build and deployment story, we want to be able to build the code automatically, and even continuously. We decide to set it up in two stages. In Jenkins each build process configuration is organized into an entity called a job. Each app that is build with Jenkins has a Job that defines the steps needed to build that app. In the first stage we schedule all jobs to run nightly. This gives us some level of build automation and gives us time to investigate merge strategies and set it up properly. The nightly job is set to run every day at midnight.



(a) The direct commit strategy



(b) The pre-tested commit strategy

Figure 5.1: Different branching strategies. A circles represents a commit and an arrow represents a reference to a commit.

5.3.1 Merge Strategy

As the name *continuous integration* suggests, code should be integrated into the mainline (or *master branch*) frequently [20]. According to Martin Fowler, frequent merges ensure that merges generally will be small and easy to perform [21]. The master branch must be stable and always in a release-ready state. If it breaks, it should be the team's first priority to fix it. A consequence of this is that the whole team is affected when a developer introduces an error, which has a negative influence on the overall productivity.

We assess two strategies to accommodate these consequences of continuous integration: A *direct commit strategy* and the *pre-tested commit strategy*. In the direct commit strategy [24], every developer integrates their code directly to the master branch. It is the developer's own responsibility that the code works. This is the simplest, and one may say, the most agile way of integrating code with the master branch. An illustration of the direct commit strategy can be seen in Figure 5.1a. A developer creates a branch from the master branch and develops their code (commits `B` and `C`) on this before merging it directly into the master branch (commit `D`).

An alternative to integrating code directly with the master branch is to use pre-tested commits [22]. A pre-tested commit uses a special branch which is an intermediate place for building and testing code before it is merged into the master branch. The code will only be merged into the master branch if it passes the tests. This ensures that the master branch will always work, but the merging workflow will be more complex as the developer must pull from one branch and push to another. The strategy is illustrated in Figure 5.1b. The developer creates a branch from the master branch and develops its code on here. When completed, the code is merged with the intermediate branch

```
1 #!/bin/bash
2 curl -s http://localhost/jenkins/git/notifyCommit?url=
      ↪ http://cs-cust06-int.cs.aau.dk/git-ro/$(basename $(pwd)) > /dev/null
3
4 echo "Thank you for your push. Jenkins will be serving you in a moment."
```

Listing 5.1: The post-receive Git hook. It is a Bash script that starts a job on Jenkins upon a push to a Git repository.

(commit *I*), which will build and test the code before eventually merging it with the master branch (commit D).

For the first sprint, we choose to implement the direct commit strategy, primarily because of its simplicity. It is important to get continuous integration up running so the developers can start to develop code, and it would be too costly to spend time implementing a pre-tested commit strategy as this will block the progress of all other developers. We are unsure about how the developers handle the increased responsibility, so in the preparation of the second sprint, we will evaluate this strategy and consider whether we should implement the pre-tested commit strategy instead.

5.3.2 Continuous Build and Deployment

We want a build job to be started automatically when changes are made to the code. We do this by setting up a post-receive Git hook that invokes Jenkins. Jenkins then automatically builds jobs that have code pushed to their master branch. The hook can be seen in Listing 5.1. `curl` sends an HTTP GET request to Jenkins with the needed information to start a build on a job associated with a particular repository. `http://cs-cust06-int.cs.aau.dk/git-ro/$(basename $(pwd))` is the link for the repository to update. Since we use the same Git hook script in all repositories, and there are several repositories, the name of the repository that invokes the hook is found by `$(basename $(pwd))`. Finally a message is displayed to the user that pushed to tell them that Jenkins will start building.

In Jenkins there is a job called *deployment* which builds all other jobs and moves the APKs to a specific directory. An APK file is the output of a build and is a package format used to distribute and install software onto Android. We remove this job and make it a part of the build process of every job to do this themselves, and ensure there are no redundant APKs present in the APK directory. By making it a part of every job we also ensure that there always is an up-to-date version of all applications whenever they pass a build, complementing our vision for continuous integration. This additional work was unplanned, and we do not have time to set it up on all jobs in this sprint.

5.3.3 Continuous Test

Now that the automatic test setup is created, we examine the different types of automatic tests we can use on Android. Because one of the ideas behind automated build is to give

the developer fast feedback on the state of the code, the build and test process should be fast. The Extreme Programming (XP) development method states 10 minutes as a guideline for how much time a build should take [10]. While we think this is a reasonable time, not all parts of the system can be thoroughly tested in that time. In such cases, Martin Fowler suggests that the fastest and general tests should be whenever a commit is pushed to the master branch, and slower tests can be triggered for later execution [20]. We call these kinds of tests *delayed tests*.

Unit Testing

During the initial investigation of the inherited code base we found some existing unit tests in one of the GIRAF apps. The unit tests runs in an emulated Android environment, or on actual Android devices if any are connected to the computer. These tests utilize the Android unit testing framework included in the Android framework [31]. We decide to postpone any further investigation into unit testing frameworks. Instead, we focus on getting the existing unit tests to run, both locally and in Jenkins. The tests were immediately runnable through Android Studio 0.4.6. However, Android Studio does something behind the scenes when it runs the tests, and therefore we cannot run them from the command line, which means that we cannot run them in Jenkins. However, the most recent Android studio, version 1.0.2, uses Gradle exclusively to run the tests. This means that when the apps are migrated to the new version, we are able to run the tests with Jenkins. The other GIRAF apps did not have any tests, so we create test projects, with examples of tests to verify that Jenkins can run the tests. This makes it easier for the other groups to start writing tests, as there is then a unit testing framework in place, as well as an example test.

Currently, Jenkins starts a new emulator for testing before each job is built. This adds a significant delay to the build. The building time when building all apps has increased from 20 minutes to 90 minutes. This is unfortunate, but for now we will not spend more time improving this. We may return to this in a later sprint. Unit tests are the simplest tests we have, and we do not want these to be handled as delayed tests. Because of the slow emulator startup, we only start a single emulator. This means that apps are tested only on a single Android device configuration, which in turn means that we may not discover all errors when testing. However, it is better than no testing and we may expand the range of devices in the future.

UI Testing and Monkey Testing

Because Android applications contain a graphical user interface, it is not sufficient to only test the backend libraries. On the overall level, there are two ways of performing automatic testing of the user interface in Android: UI testing and monkey testing. UI testing is a way of declaring specific sequences of events and their expected behaviors. For example, a test may specify a click on a settings button and assert that this actually opens the settings screen. Writing UI tests can be labor-intensive, though, and the maintenance of the tests can get quite comprehensive. As an alternative, monkey testing can be used.

Monkey testing is performed by inputting a random sequence of events, such as buttons clicks and touches, into a device. That way, the user interface of an app is randomly tested. There are no guarantees that all parts of the app will be tested, but the setup of the test is generally simple. The official Android SDK has monkey testing facilities built in, and there exists a Jenkins plugin for running these. Because of the simplicity of monkey testing and because the GUI application developers have requested this as a tool for discovering bugs in the code, this is the way we automatically test the user interface. We plan to make Jenkins run monkey tests during the night, as delayed tests, when the server is otherwise idle.

While the Android monkey tool can test multiple applications at once, the Jenkins plugin does not support that feature. The monkey test command can take as an argument a number of packages (apps) to test, using the `-p` option, for example `monkey -p package1 -p package2 -p package3`. The Jenkins plugin does not support making more than one `-p` option. The plugin is open source, so we add support for this feature, by contributing code[1]. Multiple packages are now delimited by comma. The reason for this is that we found that some users of the plugin exploited the way the package name was inserted as an argument to the monkey tool in order to insert different arguments. Because a comma character is not used in any argument to the monkey tool (compared to for example a whitespace character), we expect that no existing users will be affected by the change.

Additionally, the plugin requires the tested apps to be installed. However, we do not have the APKs of all apps easily available. Therefore we do not manage to setup monkey testing in this sprint.

Concolic Testing

Concolic testing is a way to statically analyze code in order to find bugs, e.g. potential null pointer exceptions. Concolic testing uses a combination of symbolic execution and testing of particular paths in order to maximize code coverage [62].

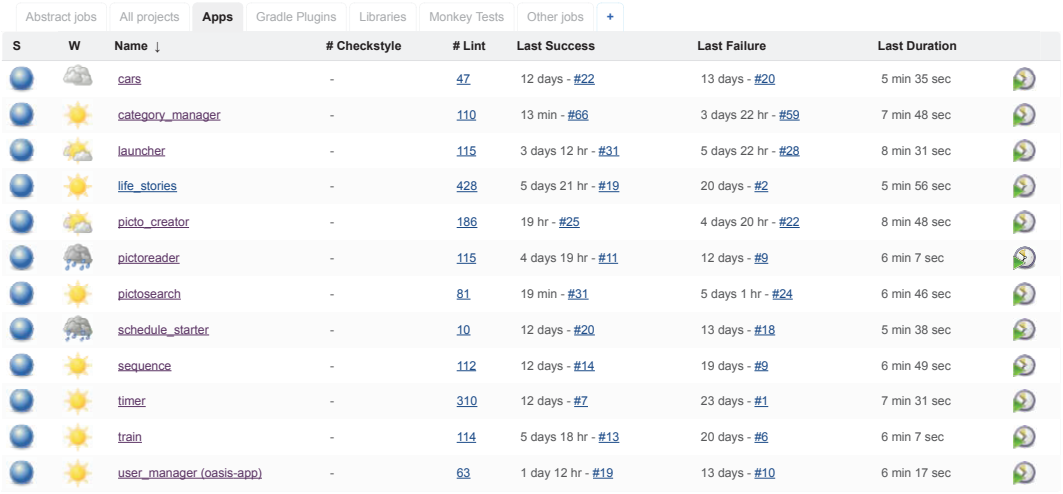
To our knowledge, the only framework for concolic testing that works with Android projects is *Acteve* [2, 1]. The documentation for this framework is non-existent, and we cannot find anything except for an unsuccessful attempt to build and run Acteve [52].

We already have unit testing and many crash reports to occupy the time of many multi-project groups, and as such we do not think concolic testing is worth the payoff, so we will not spend more time investigating this.

5.3.4 Automated Lint Check

Lint checking is static code analysis that scans the source code for potential bugs and improvements. We investigate the possibility of automating lint checks on the source code, as we suspect this will uncover a wide range of improvements.

[1] Commits on Github (`https://github.com/jenkinsci/android-emulator-plugin`): 71385cf, 811bfac, 5395bb1



S	W	Name ↓	# Checkstyle	# Lint	Last Success	Last Failure	Last Duration
		cars	-	47	12 days - #22	13 days - #20	5 min 35 sec
		category_manager	-	110	13 min - #66	3 days 22 hr - #59	7 min 48 sec
		launcher	-	115	3 days 12 hr - #31	5 days 22 hr - #28	8 min 31 sec
		life_stories	-	428	5 days 21 hr - #19	20 days - #2	5 min 56 sec
		picto_creator	-	186	19 hr - #25	4 days 20 hr - #22	8 min 48 sec
		pictoreader	-	115	4 days 19 hr - #11	12 days - #9	6 min 7 sec
		pictosearch	-	81	19 min - #31	5 days 1 hr - #24	6 min 46 sec
		schedule_starter	-	10	12 days - #20	13 days - #18	5 min 38 sec
		sequence	-	112	12 days - #14	19 days - #9	6 min 49 sec
		timer	-	310	12 days - #7	23 days - #1	7 min 31 sec
		train	-	114	5 days 18 hr - #13	20 days - #6	6 min 7 sec
		user_manager (oasis-app)	-	63	1 day 12 hr - #19	13 days - #10	6 min 17 sec

Figure 5.2: Screenshot of a section of the build overview screen, which shows the lint warning column

An official tool, Android Lint [30], for linting Android project source files exists. It checks for potential bugs and optimization improvements for correctness, security, performance, usability, accessibility, and internationalization.

There are some important considerations when choosing to lint the source code automatically. The code base is inherited from earlier years and no lint checking to our knowledge was performed on this. Linting the code will produce a considerable amount of warnings. It is therefore not possible for us to let a build fail if it contain any warnings.

We set up automated lint checking in Jenkins on all jobs. The warnings are presented in the build overview screen, as seen in Figure 5.2. We hope that having a low number of lint warnings will be incentive enough for each group to fix warnings.

Over time, we hope that the presence of serious lint warnings can be a reason to fail a build. We may adopt this practice in a later sprint, and to help speed up this adoption we may set it up as follows. As there are a large number of lint warnings already in the code, a baseline day can be selected. Groups should not be *punished* (i.e. the build fails) for lint warnings that were present before the baseline day. Only newly introduced lint warnings should be considered and punished.

5.4 Automated Documentation Generation

When working with a large code base, people will most likely not have insight in all parts of the code. Some kind of documentation is helpful in order to know how to use libraries. However the agile manifesto states that one should prioritize working code over documentation [11]. Therefore we do not intend to write comprehensible documentation of the code. A consequence of being agile is that the system architecture is likely to change rapidly — so it is important that little effort is required for updating the documentation. To make the documentation easy to find, we want the documentation

for all GIRAF projects accessible from one place. Based on these requirements, we choose to use a documentation generator which automatically generates documentation from the code rather than writing such documentation manually. The documentation will not be all-encompassing, but act more like a guide to the code.

We investigate the Javadoc [48] and Doxygen [18] tools. Both tools are cross-platform and can be integrated with Jenkins. Javadoc can generate documentation in HTML and Doxygen can generate both HTML and LaTeX. We will generate HTML documentation which will be hosted on the server.

Javadoc is the official documentation generator for Java and generates documentation based on specially formatted comments embedded in the source code. This format is integrated in Android Studio, making the documentation easily accessible where needed. In addition, parts of the existing code already contain Javadoc comments. A problem with Javadoc, however, is that it follows the packages and classes referred in a Java file. A requirement for this is that the source code must be a correctly structured Java project in order for the tool to find the different classes. This makes it complicated to comply with the requirement of having documentation of all projects in one place, because a combination of all projects may not form a valid Java project. The source code of all projects cannot simply be copied into one directory and fed to the Javadoc tool.

The Doxygen tool is a cross-language documentation generator that supports the same documentation syntax as Javadoc. Doxygen does not follow the class dependencies but simply parses the specified files. The HTML-documentation is very similar to that of Javadoc, and because it is more convenient to use, we have chosen to use this tool for documentation generation.

5.4.1 Documentation Generation in Jenkins

We have configured a Jenkins job to generate the documentation for all GIRAF projects. This job pulls the most recent state of the master branch of each project and executes the Doxygen tool on all code. On the current code base, the doxygen tool uses up to 7 minutes to generate the documentation because of the size of it. Because we do not want to block more important jobs, such as building and testing new commits, we choose to run this job nightly.

6 Sprint Review

This sprint had a number of tasks that we wanted to solve and in addition we introduced some changes to the development method. This chapter evaluates the tasks we planned and the development method followed during this sprint.

Chapter Organization This chapter is organized in the following fashion:

- In Section 6.1 we evaluate our goals for this sprint;
- in Section 6.2 we evaluate the development method in this sprint;
- in Section 6.3 we describe the results of the multi-project sprint review meeting.

6.1 Sprint Goals Evaluation

For this sprint, our main task was to setup continuous integration. The following evaluates the tasks described in Section 4.4.

Build We setup Jenkins to automatically build a GIRAF project whenever a push is made to the Git repository of that GIRAF project. The build times, however, are very slow. It can take more than the 10 minutes recommended by Beck and Andres [10] to build some jobs in Jenkins. In addition if there is a queue of multiple jobs it can take even longer. We add a user story to the product backlog to improve build times.

For integrating code with the mainline, we use a simple merge strategy which allows developers to commit directly to the master branch. The builds have generally been unstable in Jenkins. This is a potential problem because it can block developers from writing code in apps that do not work. However, because there have not been any complaints from the developers, we assume that it is not a big problem in practice. We will not work further with branching strategies.

Documentation Documentation is automatically generated each night from the entire code base and uploaded to a publicly available website. This works as required by the task.

Concolic Testing We chose not to implement concolic testing due to it being too complex to implement compared to the benefit it would give us. It may be worth looking more into this in a future sprint if the developers express an interest in this.

Lint Lint warnings and errors are automatically generated in Jenkins. We decided not to make a lint error break the build as to not punish groups for lint errors caused

by code from previous years. Lint errors are seldom critical. By displaying the lint errors on the Jenkins page some groups have indeed worked on reducing the number of lint errors. We will therefore not implement build failure for lint errors.

Unit Testing We setup unit testing such that they run as part of a build in Jenkins. When a test fails, the build fails. However, testing takes a long time because they run on an emulator which must start up as part of the test process.

UI Testing We wanted to implement monkey testing, but because the plugin in Jenkins did not support multiple app monkey testing and unavailable APKs on the server, we did not manage to implement monkey testing in Jenkins. However, we have made an improvement to the Jenkins plugin which makes it easier to eventually implement monkey testing in a future sprint.

6.2 Process Evaluation During Sprint

During sprint 1 we changed a few things in the development method of the multi-project:

Customer Relations Role The *customer relations* role was initially vague in its description. Some understood it as being a product owner (PO) role, while others understood it only as a contact person for the external customers. We solve this by letting the role be the contact person for the external customers and assigning the PO role to other groups as explained below.

Multiple Product Owners and Meetings Originally, a single sprint review meeting with the external customers was planned. Due to concerns from the semester coordinator, it was changed into several meetings. Last year, there was one sprint review meeting, and it was observed that the external customers were not interested in most of the progress the DB and B&D groups had made. Therefore we decided that the external customers should only attend a meeting where the GUI groups would present their progress as well as the relevant progress from DB and B&D (if any).

We seek a way of organizing the different teams in a structure that allows them to work with user stories relevant to their subprojects. Bird and Davies [13] describes four methods of organizing teams in Scrum of Scrums:

1. Independent groups with a customer and PO of their own. They also have their own backlog.
2. Groups have a single common backlog and share the customer and PO
3. Groups have their own customer and PO, and a backlog that takes items from a large common backlog
4. Subproject has their own PO, customer, and backlog, and groups can serve as customers for other groups

The first and second options are the simplest, but they can prove difficult in the multi-project, as groups from each subproject may be dependent on the other

groups. For example, the GUI subproject may require changes to the database structure, which should be resolved by the DB subproject. The third option adds an extra layer on top of the previous described methods, which allows user stories from the external customer to be distributed among the individual groups. However, it still has the issue that the primary customer is the external customer. The final option, on the other hand, allows for example the GUI groups to be customers of the B&D groups. Thus, the external customer can be abstracted from internal needs and technicalities. This structure fits the needs for the multi-project well. Specifically, there are three POs in the multi-project, one for each subproject. The customer for the GUI PO is the external customer, whereas the customers for the DB and B&D POs are the groups from the other subprojects.

We choose to present the fourth method to the groups of the multi-project who approve it. The PO roles are designated to a group (and not a single person) in each subproject. This differ from the regular Scrum method, where a product owner is a single person.

The result of this is the structure described in Section 2.1, where there are three POs, three sprint review meetings, and three sprint planning meetings.

Office Hours In the beginning of the multi-project, it was decided that all groups must be present in their group rooms from 09:00–15:00. However, after the decision had been made, some groups indicated that they did not intend to follow this requirement. This meant that other groups complained about not being able to do their work efficiently due to dependencies on the groups that were not present. Thus we were asked to create a discussion about this at the weekly meeting in the multi-project. It is decided to relax the office hour requirement, such that it is acceptable to not be physically present in one's group room — instead each group has to be reachable by e-mail in the same time period. Groups are, however, encouraged to be physically present. This way all groups agree on the office hours.

6.3 Multi-Project Sprint Review

In the end of this sprint, a final common meeting among all groups is held. The purpose of this meeting is to evaluate and reflect upon the decisions in this sprint and to evaluate the process.

It is suggested to add one or two days between the GUI sprint planning and the DB and B&D sprint plannings. Since the GUI groups have to process user story prioritization done by the external customers as well as incorporate new requirements into the product backlog, they need time to do so, before they can elicit derived requirements to the DB and B&D groups.

A grace period after the sprint planning meetings is also suggested, to allow user stories to be incorporated a few days into the sprint. We understand this is not strictly Scrum, however it is necessary because, for example, a DB user story might require a new user story in GUI or B&D to be selected. If sprints were shorter, the grace period

may not have been as necessary as it is now, since the user story could be selected shortly after. But when having one month sprints much time will be wasted just because the user story could not be selected.

Furthermore, it is suggested that the three product backlogs are merged into one document. This makes it easier to access the product backlog and also hopefully decreases overlap.

At last, the server was rebooted the day before the GUI sprint review meeting. Unfortunately, the server did not boot correctly, and the server was not working. This posed a problem, since it made it difficult to show the external customers the product. It is suggested that the server should not be rebooted just up to a sprint review.

Sprint 2

7 Sprint Planning

This is the first sprint in which each subproject holds a separate sprint planning meeting. We attend the public parts of all sprint planning meetings, and of course also the internal part of the sprint planning meeting of our subproject. Our help is requested for the DB sprint planning process.

Chapter Organization This chapter is organized in the following fashion:

- In Section 7.1 we elaborate on the help we provide for the sprint planning of the DB subproject;
- in Section 7.2 we describe the result, i.e. the selected user stories, of the B&D sprint planning meeting;
- in Section 7.3 we describe our own group sprint planning and lists our tasks with reference to a user story.

7.1 DB Sprint Planning

While we attended the first part of the DB sprint planning meeting as customers, we are asked to attend the second part as well, which is normally reserved for only the DB groups. We are asked since the initial part of the sprint planning meeting did no go so well — too much time was spent discussing irrelevant things, since the product backlog was "too close" to external customers' needs, rather than internal user needs. In addition the DB POs are not entirely confident in running parts of the meeting.

At the beginning of the second part of the meeting, we clarify how the remaining part is going to go: Groups are to pick user stories for this sprint, so that they have enough work for the sprint. They can estimate the user stories at the meeting, or they can do so separately and report back to the PO. The user stories are not altered to be "developer friendly" at the meeting, since there is no time for that. Instead we suggest that the POs do this for the next sprint planning. We do not influence the user stories they pick for their release backlog.

7.2 B&D Sprint Planning

At the B&D sprint planning we choose to work on the following user stories in this sprint. They are labeled with a number in parentheses for reference.

Auto Upload Alpha and Beta Releases of Apps (1) Apps should automatically be uploaded to the Google Play alpha channel when they build successfully. They must be uploaded to a beta or release channel at the end of a sprint.

Faster Build (2) Builds are slow. Builds should be sped up — by how much is not specified.

Set up Monkey Testing (3) Monkey testing for each app should be run automatically.

Test Case Installation (4) A test that automatically installs all apps on a single device. At the end of sprint 1 the GUI groups had issues installing some apps on a device when other apps were already installed due to a conflict. To avoid this situation in future sprints, the GUI groups requested a test to catch such errors.

Code Coverage (5) Jobs in Jenkins should provide code coverage metrics for the tested code.

7.3 Group Sprint Planning

At our internal sprint planning we divide the chosen user stories into tasks and estimate those. For this sprint we have a total of 58 half days of work. Table 7.1 shows the tasks we have for this sprint. Estimations in parentheses are the result of estimations being updated. Tasks with a plus (+) are tasks that are added during the sprint as they are discovered.

We made tasks for finishing the report of the previous sprint estimated for 30 half days. The original total amount of work estimated for both the report and user stories is 62. While this is 4 units more of work than we have available, many of the report tasks are estimated as being 1, but in fact take less time. Therefore an additional 4 units of work is acceptable.

The task *make Jenkins job for beta releases* was also rejected during the sprint, as we discovered there was no need to fulfill this task. The Google Play group can easily "upgrade" an alpha release to a beta release or actual release.

Task	User Story	Estimation
Make Gradle plugin for signing APKs	1	2
Remove versionCode incrementation from debug Gradle task	1	2
Upload newest APKs to Google Play as alpha	1	1
Put Gradle plugins in Jenkins (+)	1 & 2	1
Publish AAR files to Artifactory automatically from Jenkins (+)	2	6
Put metadata-lib in Jenkins (+)	2	1
Make Gradle download binaries from Artifactory (+)	2	2
Set up Artifactory	2	1
Move APKs to ftp server	3	1
Make test that installs all available APKs	4	1
CoCo code coverage	5	8
Down-prioritized tasks		
Investigate non-emulator testing	2	6 (1)
Investigate tablet slaves for testing	2	7 (1)
Missed tasks		
Make monkey test job in Jenkins for each app	3	2
Rejected tasks		
Make Jenkins job for beta releases	1	2
Original total		32
Total		30

Table 7.1: Sprint backlog for sprint 2, excluding report tasks. The tasks are listed in no particular order.

8 Improving Automated Build

For this sprint we choose to work on four user stories that we expect to improve automated building.

Chapter Organization This chapter is organized in the following fashion:

- In Section 8.1 we explain how we restructure jobs in Jenkins to ease further configuration needs;
- in Section 8.2 we enable generation of a code coverage report and show the result in the overview list in Jenkins;
- in Section 8.3 we automatically upload successful builds to the Google Play alpha channel;
- in Section 8.4 we describe our work towards making monkey testing work;
- in Section 8.5 we set up a test case that installs all apps on the same device to check if apps are compatible, since this has been an issue.

8.1 Restructuring Jenkins

The setup of Jenkins used in sprint 1 was tedious to work with since all jobs were configured independently. If a change had to be made to several jobs, we would have to manually configure the change in each job. Not only did this take a considerable amount of time, but it was also prone to human errors during the process. In this sprint, we predict that we will have make small modifications to all jobs several times. This would be very tedious, so we decide to make job configurations easier to manage, even though it is not connected directly to a story. We consider it refactoring.

The jobs in Jenkins are generally very similar in their configuration. We can benefit from having a base configuration, which the jobs only modify. There exists a Jenkins plugin for this, called inheritance-plugin [57]. With this plugin installed, whenever we decide to make a change to e.g. the build system, it will not be necessary to change this in each job. Instead, the change can be made on the base job, and all relevant jobs will inherit this change. This also ensures that jobs follow a consistent pipeline and thus do not differ from job to job.

The plugin requires the jobs to be of a special *inheritable* type. We therefore have to convert the existing jobs to inheritable jobs to take advantage of this. As an existing job cannot automatically be converted to the inheritable type, we have to re-create all the jobs. We do not consider this a concern, as the time to setup will be considerably shorter when taking advantage of the inheritable job type. When the old jobs are removed the

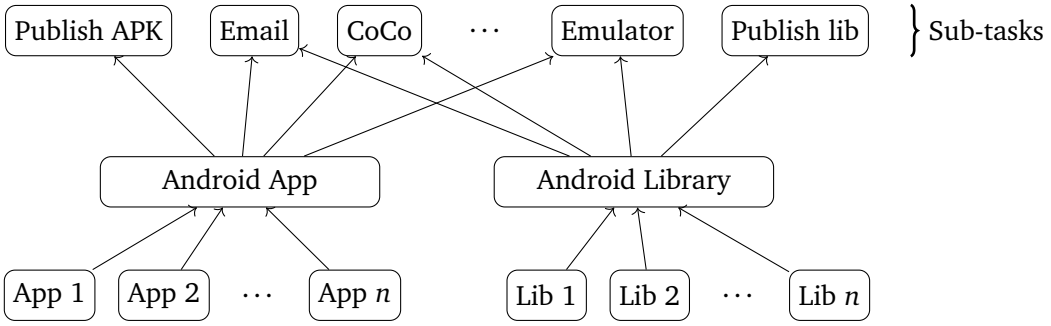


Figure 8.1: Jenkins inheritable jobs

build history will be lost. This is a minor nuisance but we consider it a small price to pay, compared to the advantages. When deciding how to structure the build, we see two general categories that are sufficiently distinct: *Android Apps* and *Android Libraries*. We create an abstract job for each of these. They do overlap somewhat in functionality, which means we have to create an abstract job for each build step. As such we create abstract jobs for e.g. *Email*, *Emulator*, and *CoCo* (code coverage). The abstract jobs Android App and Android Library inherit from the small abstract jobs that are relevant. We have modeled this as a diagram in Figure 8.1.

8.2 Code Coverage Reports

We have a user story which states that Jenkins should provide code coverage metrics for every GIRAF project. This user story was suggested by a group who was writing tests for a database project and wanted a measure of their progress. Their main request is for a percentage of lines of code covered. A tool for code coverage report must at least provide this metric. In version 0.10 of the *New Android SDK Build System* [4], support for the JaCoCo [19] Java Code Coverage Library was included. It meets all of our demands for metrics and is nicely integrated into the Android and Gradle build system. To make JaCoCo generate a report, we simply enable it in the `build.gradle` file of the project.

Now we generate code coverage reports for debug builds locally and in Jenkins. We would also like to publish the code coverage results in Jenkins. There exists a Jenkins plugin [37] for this purpose. This plugin is easy to setup, provides detailed coverage statistics, and an overview with the percentage of lines of code covered. We use this plugin for publishing the code coverage metrics in Jenkins. An example of the Jenkins overview screen with code coverage metrics can be seen in Figure 8.2.

8.3 Upload of Apps to Google Play

Jenkins compiles and builds the GIRAF projects but does nothing with the generated APKs. One of the user stories selected in this sprint is *automatic upload of alpha releases to Google Play*.

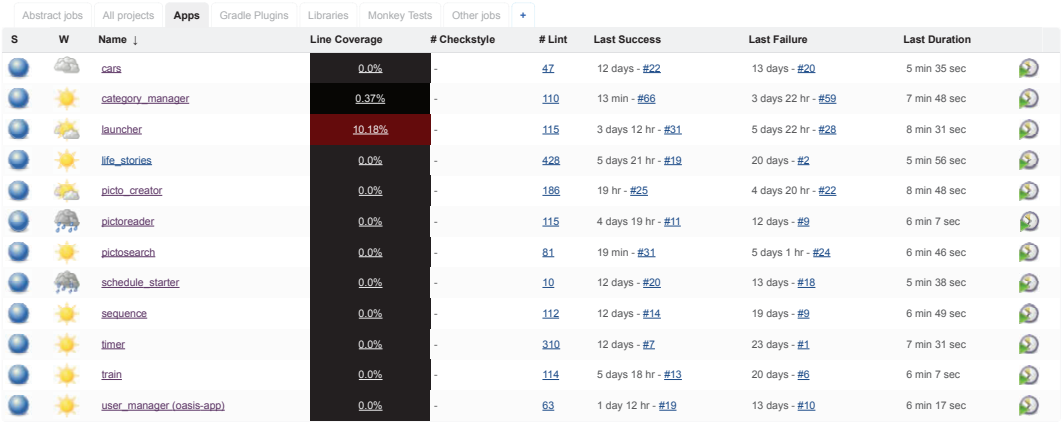


Figure 8.2: Screenshot of a section of the build overview screen which shows the code coverage column

Before the APKs can be published to Google Play, they need to be signed with a signature. The Android plugin for Gradle has functionality for automatic signing of APKs, and we will use this to sign. A keystore file is used to sign, and we are not interested in everybody having this file as it serves as a proof of identification. We save the file on the server, which means that it becomes impossible to build release versions of the apps locally. When uploading a new app to Google Play, the version code of the new app must be greater than the version code of the app already in the app store. Incrementing the version code is not done per default, so we need to set up the build to increase the version code every time an app is successfully built.

We have written a Gradle plugin handling this with the major part of this seen in Listing 8.1. The full plugin can be seen in Appendix B. The plugin keeps a `.properties` file with the current version code for all apps. Upon executing the `increaseVersionCode` Gradle task, it reads the version code (line 8) from the properties file and increments it (line 12). Then it writes it to the app manifest file, such that the build following will use the new version code. Finally, it updates the property file with the new version code, such that the next build will use this version code. If no current version code is found in the properties file, we assume the app is new and starts at version code `1`, and as such we do not increment it.

Now that the APKs have been signed, they need to be uploaded to Google Play. This can be done in two ways: Via a Jenkins plugin [50] or through Gradle [12]. The Jenkins plugin is easy to use, but requires that the exact name and location of the APK to upload is known. The Gradle plugin knows this already, since the information is already present in the Gradle build environment. Therefore we use Gradle to publish to Google Play via a Google Play API. After the APKs have been uploaded, they are moved to the FTP server which is hosted on the same machine as Jenkins in the directory `/srv/ftp/`. This way the APKs are also available outside of the Google Play store.

```
project.task('increaseVersionCode') << {
    // [...]
    // Check to see if properties file exists.
    if (project.file(versionCodesFilePath).exists() != true) {
        throw new GradleException("No version code file found. Only Jenkins should
    ↪ run this task")
    }
    // [...]
    def newVersion = getVersionCode(project, applicationId)
    def versionCode = newVersion['value']
    if (!newVersion['created']) {
        // Increment version code if not new
        versionCode++
        // Write incremented version code to manifest
        // [...]
        // Write incremented version code back to properties file
        // [...]
    }
}

def getVersionCode(project, applicationId) {
    // Reads the version code from the properties file. Returns 1 if the version code
    ↪ does not exist.
}
```

Listing 8.1: Part of our Gradle plugin for updating version code (written in Groovy)

8.4 Monkey Testing

During sprint 1 we encountered some difficulties related to monkey testing. In order to monkey test apps we need to install those apps on an Android device, but we did not have these apps easily available. As described in Section 8.3 the signed APKs for each app is now stored on the FTP sever. As the FTP server is hosted on the same machine as Jenkins, we can directly access the APKs from the directory. To install we simply need to identify the newest build of each app, as, for now, the old builds are kept as well.

To find the APK for the newest version of a specific app, we use the script seen in Listing 8.2, parsing to it the application id of the app. All APKs containing the given application id is then found. This result is piped to `sed 's/.*b//' | sed 's/_release_aligned.apk//'` that removes everything but the build number of the files. `awk '$0>x{x=$0};END{print x}'` [40] then finds the maximum build number. Finally the path of the newest APK for the given application id is found.

For example, consider the following APK files:

```
dk.aau.cs.giraf.launcher_v2.3b18_release_aligned.apk
dk.aau.cs.giraf.launcher_v2.3b19_release_aligned.apk
dk.aau.cs.giraf.launcher_v2.3b20_release_aligned.apk
dk.aau.cs.giraf.launcher_v2.3b21_release_aligned.apk
```

The APK with the highest build number is the newest. The build number is the number after the `b` in the file name. Running the script in Listing 8.2 with the application id

```
1 #!/bin/bash
2 NEWEST_BUILD="$(find /srv/ftp/newest_apks/ -name "$1*release_aligned.apk" | sed
      ↪ 's/.*b//' | sed 's/_release_aligned.apk//' | awk '$0>x{x=$0};END{print x}')"
3 find /srv/ftp/newest_apks/ -name "$1*b${NEWEST_BUILD}_release_aligned.apk"
```

Listing 8.2: Bash script that finds the newest APK for a particular application id

```
1 #!/bin/bash
2 PACKAGE_NAMES="$(find /srv/ftp/newest_apks/ -name "*release_aligned.apk" | sed
      ↪ 's:.*/::' | sed 's:_v.*::' | sort | uniq)"
3
4 for p in $PACKAGE_NAMES
5 do
6     /srv/scripts/find_newest_apk.sh $p
7 done
```

Listing 8.3: Bash script that finds the newest available APK for all apps

`dk.aau.cs.giraf.launcher` will find the APK in line 4 (with version number `v2.3b21`).

When the app has been installed on the device, we run a monkey test on it using the Jenkins plugin. Starting a monkey test will launch the app. This poses problems for the majority of the GIRAF apps, as they requires extra information when starting, such as user information. If the apps are not given this information at launch they will crash. This means that all monkey tests report failure. The only app that require no extra information is the Launcher app. However, as this app uses the first 5–10 minutes downloading pictograms, the monkey only tests the loading screen.

The `monkey` command does not support sending extra information when starting apps. We did not anticipate this obstacle and did therefore not manage to implement monkey testing for apps fully.

8.5 App Installation Test Case

To ensure that all apps of the multi-project are compatible, a test in Jenkins is run nightly that installs all apps on a single device. At the end of sprint 1 the GUI groups had issues installing a combination of apps on a single device — this test case should avoid such an issue in the future.

To install the newest version of all, we have to find the APKs on the server. The script seen in Listing 8.3 finds the application id for each available app by finding all files in the directory containing APKs. It then pipes these files to `sed 's:.*/::'` which removes the path of the file. This is then piped to `sed 's:_v.*::'` that removes everything following the package id. The output of that is then sorted and all duplicates are removed using `sort | uniq`. The sort is necessary as `uniq` only removes duplicate lines that are adjacent.

The application names are then sent to `find_newest_apk.sh`, as seen previously in Listing 8.2.

When the newest available APK for each app has been found, they are installed on an

```
1  #!/bin/bash
2  for a in $@
3  do
4      INSTALL_OUTPUT="$($ANDROID_HOME/platform-tools/adb install $a)"
5      echo $INSTALL_OUTPUT
6
7      IS_SUCCESS="$(echo "$INSTALL_OUTPUT" | grep -i success)"
8
9      # Check if output message does not contain success, since adb install does not
       ↪ provide exit code != 0 at failure
10     if ! [ "$IS_SUCCESS" ]; then
11         echo "Error installing $a"
12         exit 1
13     fi
14 done
```

Listing 8.4: Bash script that installs the given APKs to an Android device

Android device. This script is seen in Listing 8.4. Line 4 tries to install an APK and stores the output in `INSTALL_OUTPUT`. The output is stored, as `adb install` gives an error code of `0`, even if it fails. We therefore have to read the output to check if an error occurred so that Jenkins will report the job as a failure. This is done at line 7 where we check that the output contains the string `success`. The `i` option makes `grep` case insensitive. At lines 10–11 the script exits with `1` if the output does not contain `success`. If the installation of any app fails on the device, the Jenkins job will fail.

9 Improving Build Times

The multi-project groups want the build times in Jenkins to be faster. When multiple changes are pushed to the master branch within a short space of time, the triggered build jobs will congest the build queue. This means that even though the build itself may take only a few minutes, the time it takes from push to finished build may be much longer and exceed the 10 minutes advocated by Beck and Andres [10].

Chapter Organization This chapter is organized in the following fashion:

- In Section 9.1 we measure the time of building the Launcher app to understand which areas of the build process to make faster;
- in Section 9.2 we present an alternative solution to the current Git submodule dependency system;
- in Section 9.3 we choose a dependency repository for storing binaries;
- in Section 9.4 we define the workflow for the multi-project developers when working with the new dependency system;
- in Section 9.5 we configure the GIRAF projects to use the new dependency system;
- in Section 9.6 we describe how versions of binaries are automatically managed;
- in Section 9.7 we describe a serious bug in the Android Emulator plugin for Jenkins and present a solution;
- in Section 9.8 we evaluate the build times after implementing binary dependencies;
- in Section 9.9 we discuss the opportunity to improve build times further by implementing non-emulator testing.

9.1 Measuring Build Times

We measure the build times of the different parts of the Launcher project in order to identify which parts of the build process to make faster. The Launcher project is the main application which depends on many other subprojects (henceforth a “dependency”). These dependencies are managed as Git submodules [27], which basically clones the contents of a repository (the dependency) at a specific revision into another repository (the dependent project). The build timings are measured on the Jenkins server and shown in Figure 9.1 (slow emulator). This shows the build timings of the Launcher application and the dependencies Oasis-lib, Giraf-component, Local-db, Barcode-scanner, and Metadata, as well as the start up time of the emulator. Notice that the Metadata

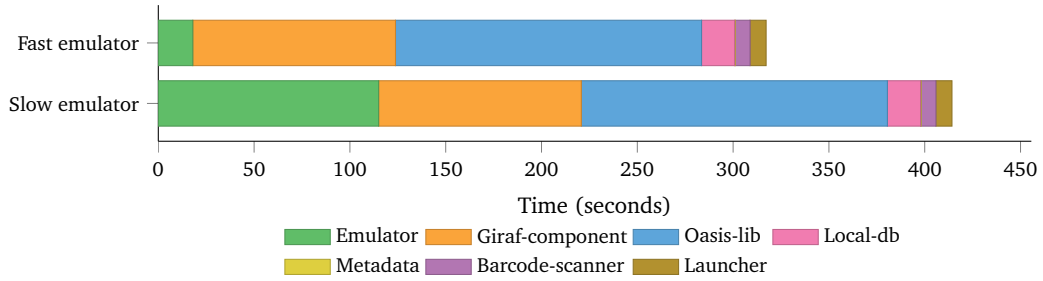


Figure 9.1: Timings during build of the Launcher project before and after updating emulator plugin. The Metadata dependency takes so little time that it cannot be seen.

dependency takes so little time that it cannot be seen. As can be seen, the emulator, the Giraf-component, and the Oasis-lib projects have the most significant influence on build times. These three parts use about 90 % of the total build time. The actual Launcher application itself takes only very little part of the total build time.

To understand which parts of the dependencies that take time, we measure the individual build steps (tasks) performed during a build. These can be seen in Appendix A. The reason that the Oasis-lib dependency takes a long time to build is that it contains a great number of tests which run on the emulator. These tests are run every time a project dependent on Oasis-lib builds — even though the library itself has not been updated. The most significant task when building the Giraf-component library is the test task as well. However, the Giraf-component library only contains one simple test, so we do not expect the actual test execution time to use much of this time. Instead, we expect it to be caused by the order in which projects are tested. When preparing the tests, the tests from all projects are combined into a single APK. The first test task run is responsible for installing this application on the emulator, while the subsequent test tasks can skip this step. Because the Giraf-components project is the first in the sequence of libraries to be tested, this task also installs the tests on the emulator which is likely to take some time.

As a preparation step, we first update the Android Emulator Jenkins plugin [49] to a new version with improved emulator stability. We do this to ensure that we do not work on improving parts which are already improved in the most recent version of the plugin. After updating, we measure the build times again. As shown in Figure 9.1 (fast emulator), the emulator startup time is significantly decreased. Because of this, the emulator startup time is no longer the primary concern, and we focus on decreasing the dependency build times.

One obvious way of decreasing overall build times is to use faster hardware on the server. However, because the multi-project has no financial income, buying new hardware is generally the last resort. In addition, it is difficult to estimate how much this will improve the build times in practice. Another option is to not test dependencies when dependent projects are built, but only when changes happen in the actual dependency project. While this will decrease the overall build times, each dependency is still built every time a dependent project is built, which limits the amount by which we can decrease

the total build times. Not testing each build of dependencies, however, is not favorable. Because of this, we look at improving the build times in a different way, specifically by using pre-compiled libraries.

9.2 Dependency Management

Instead of building and testing dependencies each time a dependent project is built, we look at referencing pre-compiled and pre-tested library files. We are inspired by the way `.jar`-files are typically used as libraries for Java projects. This way of managing libraries has a number of advantages compared to the current setup:

Pre-Compiled and Pre-Tested Libraries are binary pre-compiled and pre-tested files. This means that dependent projects do not have to build and test all dependencies, which decreases the build time significantly.

Quality Control Libraries are built and released only if the tests pass, so there will never be dependencies which cannot compile or do not work[1].

Integrity All libraries are built on the same machine and build environment, which means that no machine-specific configurations will influence the released libraries [33, 34].

Cleaner Structure Nested dependencies (dependency A depends on dependency B which depends on dependency C) is well defined and nicely handled. The individual libraries do not include their dependencies, so it is always the dependent library that has to include all dependencies. Currently, the same dependency may be included multiple times in the same project.

The Android counterpart of Java `.jar`-files is `.aar`-files. These files are similar to `.jar`-files, except that they can contain Android-specific dependencies such as icon resources as well [3].

With this solution, it is clear that we no longer want to use submodules for managing dependencies. Coincidentally, the Git responsible group works on a user story which states *Remove Git submodules*. The reason for this user story is that the developers generally find the submodules difficult to work with. Because this overlap with our solution to making build times faster, we decide to solve this in collaboration.

An important requirement for the solution, besides removing Git submodules and improving build times, is that every build should be reproducible. This means that we must be able to reproduce any previously released version. This is a very important part of continuous integration as it makes it possible to reproduce bugs from previous versions [23, 33]. In the current way of managing dependencies, this requirement is fulfilled.

[1] Of course, there may be bugs even if all tests pass, but good test cases decrease the risk

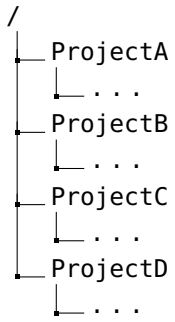


Figure 9.2: Single repository file structure

9.2.1 Agreement Upon Submodule Replacement Strategy

The Git responsible group initially proposes to remove submodules by merging all repositories into one, each GIRAF project having its own directory located in the root directory of the repository, as shown in Figure 9.2. Dependencies are then handled by referring to the relative path of the dependency. For example, if `ProjectA` depends on `ProjectB`, `ProjectA` will contain a reference to `../ProjectB`. This solution makes all releases reproducible, which is one of the requirements to the solution. The repository can simply be checked out to a specific commit corresponding to a specific release. A problem with this, however, is that the structure does not use pre-compiled libraries and as such it does not decrease build times as much as we want. Each dependency is built every time the dependent project is built.

An additional problem is that all projects are forced to use the most recent versions of its dependencies. As soon as a dependency is updated, it is automatically applied in all dependent projects. If the interface to a library changes without keeping backward compatibility, all dependent projects will stop working.

Instead, we need a structure which allows projects to reference their dependencies as pre-built libraries while being able to reproduce any build in history. For inspiration, we look at how dependency management is done in the five most starred Android projects on Github[2]. In all of these projects, dependencies are handled by Gradle and Maven. Maven is a build tool similar to Gradle, but its dependency management mechanisms are supported in many build tools, including Gradle. Dependencies are declared in the build configuration file, for example in a Gradle file, each of which uniquely identified with a group (package), name, and version number. Listing 9.1 shows how a dependency (the *gson* library) is declared in a Gradle file. When the project is built, the build system automatically searches for the dependencies on the declared repositories and downloads them. Because the version code (`2.0` in this case) is used to identify dependencies, and the build file is under version control, it is possible reproduce builds.

[2]As of April 15th, these are *ioshed*, *ViewPagerIndicator*, *retrofit*, *EventBus*, and *PhotoView*.

```
repositories {
  mavenCentral()
}
dependencies {
  compile 'com.google.code.gson:gson:2.0'
}
```

Listing 9.1: Dependency declaration in Gradle

9.3 Dependency Repository

By hosting our own internal Maven artifact repository for our libraries, we are able to declare the dependencies in the build-file and at the same time remove Git submodules and make builds happen faster. This means that we use separate repositories for source code and binary files. The main reason for this is that Git does not handle binary files well. It keeps every version of binary files [25], which will use much space over time. In an artifact repository, on the other hand, only release versions of libraries and a configurable number of development (pre-release) versions are saved.

Different pieces of software for managing artifact repositories exist, which we will present in this section. We restrict ourselves to looking at free repository systems supported by Gradle [32]: *Apache Archiva* [9], *Sonatype Nexus* [59], and *Artifactory* [36]. Overall, the only requirements for an artifact repository management system are:

Repository Browser It should be possible to browse the artifacts in the repositories, so that the developers can identify the most recent versions of the libraries.

Role Management Every developer should be able to browse the repository, but only specific people should be able to configure the repositories.

Each of the three repository systems comply with these requirements. They can all be browsed and configured through a web server. As such, it does not really matter which system we choose. We choose to use Artifactory because it has a plugin for Jenkins which may be useful for future projects. We install the software on our server and do the actual submodule replacement in collaboration with the Git responsible group.

9.4 Dependency Workflow

Because of the new way of managing dependencies, all developers have to change their workflows. To make the transition run smooth, we define a workflow for the new structure, which defines the following tasks:

Release Management and Versioning During development, libraries should be released as work-in-progress artifacts. When they are stable, new versions should be released.

Declaring Dependencies It should be clear for the developers how to declare dependencies and their versions.

Local Development Developers should be able to test changes to libraries in dependent projects without committing them to the master branch.

We demonstrate this workflow to the multi-project groups at the sprint review, so that they know how to work with the new dependency system. We also write a guide which they can use for future reference. The guide is written in Danish and can be seen in Appendix C.

9.4.1 Release Management and Versioning

With the introduction of artifacts, we now distinguish release versions from pre-release versions. In addition, a release can be a *major*, *minor*, or *patch* release. The developers need some guidelines to know when to release the different kind of releases. Rather than inventing our own convention, we rely on the Semantic Versioning 2.0.0 specification [51], which has been adopted in several Apache projects [6, 7, 5]. Because the versioning is clearly specified, we expect to avoid potential problems caused by libraries breaking backward compatibility. In summary, the Semantic Versioning 2.0.0 specification states that the version name is of the form `major.minor.patch`, each of which being a non-negative integer. For example, version `2.5.1` has major version 2, minor version 5, and patch version 1. The individual numbers are increased as new versions are released. To summarize the Semantic Versioning 2.0.0 specification, the numbers are increased according to the following:

Major When changes without backward compatibility are introduced, it should be released as a new major version. The minor and patch numbers are reset to 0.

Minor Minor releases are used when new functionality is added with backward compatibility. The major version is unchanged and the patch version is reset to 0.

Patch A patch is for introducing changes without new functionality (typically bug fixes). The major and minor versions are left unchanged.

During development, pre-release versions can be denoted by the version code followed by a hyphen and an identifier (e.g. `2.5.1-ALPHA` or `2.5.1-SNAPSHOT`). Due to the specifics of Maven dependencies, we require this identifier to always be `SNAPSHOT`. This enables Maven to handle pre-releases differently than release versions.

9.4.2 Declaring Dependencies

The new pre-built dependencies are declared almost the same way as previously. Instead of referring to a local directory containing the dependency, the reference is declared as a Maven dependency (as previously described). To accommodate this, we release the current version of each library as version 1.0.0 and make all dependent projects reference these. Listing 9.2 shows an example of how the dependencies are declared for a GIRAF project, including the declaration of the repository containing the dependencies.

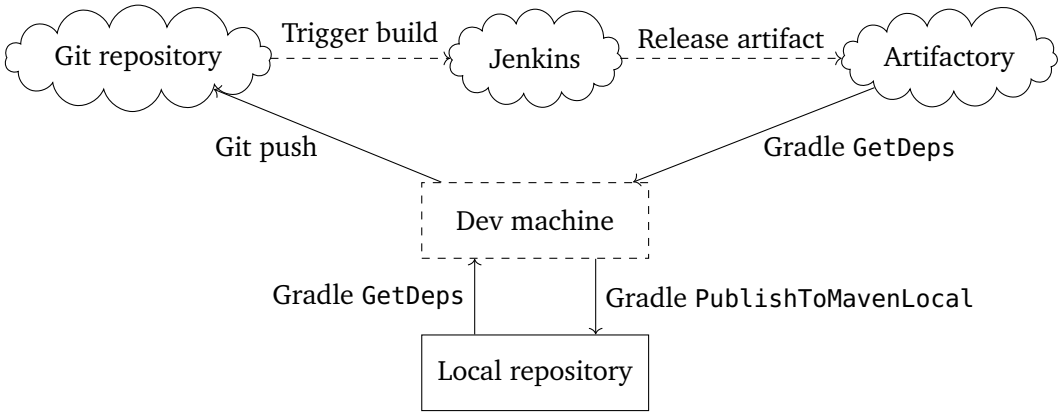


Figure 9.3: Overview of dependency management workflow. A cloud represents a server application. Filled arrows represent actions initiated by the developer, and dashed lines represent actions initiated by a server applications. The direction of arrows indicate in which direction data (source code or binary data) is transmitted.

```
1  repositories {
2      maven {
3          url 'http://cs-cust06-int.cs.aau.dk/artifactory/libraries'
4      }
5      mavenLocal()
6  }
7
8  dependencies {
9      compile(group: 'dk.aau.cs.giraf', name: 'girafComponent', version:
       ↪ '1.3.0-SNAPSHOT', ext: 'aar')
10     compile(group: 'dk.aau.cs.giraf', name: 'oasisLib', version: '1.1.2', ext: 'aar')
11 }
```

Listing 9.2: Dependency declaration for a GIRAF project

9.4.3 Local Development

Developers must be able to test their changes in the dependent projects locally without committing their changes, as part of continuous integration [20]. Our solution is to publish the dependency on a local Maven repository on every development machine. To test the dependency in another project, the dependent project points to the local repository rather than that on the common repository. Conveniently, the Maven plugin for Gradle has support for publishing and referring to local repositories. The build task `publishToMavenLocal` publishes an artifact to the local repository.

Figure 9.3 shows an overview of the dependency management workflow, from local development to releasing artifacts on Artifactory.

9.5 Configuring the Projects to Publish

Each library project has to be configured to publish binary files. This configuration is set in the build files. Instead of having to repeat the same information in the build file for every library project, we make a Gradle plugin which defines all properties needed for publishing, called the `giraf-lib-gradle-plugin`. The `apply` method of the plugin, which is invoked when the plugin is applied in the build file, can be seen in Listing 9.3. The full code can be seen in Appendix B.

When a build is started in Jenkins, the `versionName` property is set to the version of the library to release, e.g. `5.2.3`. When a library is built locally, this property is not set. The plugin therefore sets the version name to `0.0-SNAPSHOT` (lines 9–10). In line 16, the plugin checks if a file containing login credentials for Artifactory exists and then loads it in line 17. The repository to publish to is then configured in lines 19–30.

Because the credentials file is stored on the build server, it is also not possible for the developers to publish libraries to Artifactory themselves; each build must pass in Jenkins before it is released.

9.6 Automating Dependency Management

With the dependency workflow defined, we integrate it with continuous integration. As earlier mentioned, we want to automate as much as possible to ease the work of the developers. For dependency management, Gradle already does a lot for us: It automatically locates and downloads declared dependencies and creates tasks for publishing new ones to the artifact repository.

In addition to this, we automate release versioning to some degree. Developers need not to declare the version of a library in its build file — instead, they simply include the keywords `@patch`, `@minor`, or `@major` in the commit message to release a patch, minor, or major version, respectively. This automatically takes care of increasing and resetting version numbers. For example, if the current version is `1.2.3`, a `@minor` keyword will release a new artifact with version `1.3.0`. If no keyword is given, the release is assumed to be a pre-release, which causes the version name to be the *next* patch release number with a `-SNAPSHOT` postfix (e.g. `1.2.4-SNAPSHOT` if the most recent release is `1.2.3`). If multiple commits are pushed at the same time, we trigger only one pre-release build per release. For example, if it contains four commits with messages `test1`, `test2`, `@patch`, `test3`, respectively, two pre-releases (one before and one after the patch) and one patch are built. Similarly, if a push contains several commits without any release keyword, only one pre-release will be built. Figure 9.4 illustrates this behavior. The Git hook previously described is extended with functionality to manage this. We implement this automation in collaboration with the Git responsible group. As part of the implementation phase, we refactor by changing the programming language from Bash to Python, because of the greater level of abstraction provided by Python compared to Bash. Part of the code is shown in Listing 9.4. This function is called for each commit message contained in the push. It first identifies the library corresponding to the repository pushed to (lines 13–14),

```
class LibraryPlugin implements Plugin<Project> {
  // ...

  void apply(Project project) {
    // Create giraf plugin extension
    GirafLibrary girafLib = project.extensions.create ('girafLibrary', GirafLibrary)

    // Set version name
    if(!project.hasProperty('versionName')) {
      project.ext.set('versionName', '0.0-SNAPSHOT')
    }
    // Apply plugins needed for libraries
    project.afterEvaluate {
      // Read artifactory credentials
      Properties props = new Properties()
      if (project.rootProject.file(artifactCredentialsPath).exists()) {
        props.load(new
    ↪ FileInputStream(project.rootProject.file(artifactCredentialsPath)))

        project.publishing {
          publications {
            repositories.maven {
              if(project.versionName.endsWith('-SNAPSHOT')) {
                url artifactUrlSnapshots
              } else {
                url artifactUrlReleases
              }
              credentials {
                username props['username']
                password props['password']
              }
            }
          }
        }
      }

      // ...
    }
  }
}
```

Listing 9.3: Part of Gradle Plugin to publish libraries (in Groovy language)

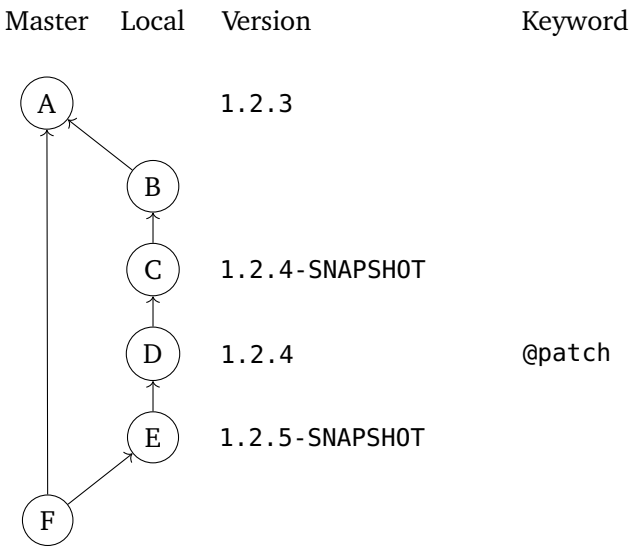


Figure 9.4: Behavior of version codes for libraries. The circles represent commits. The Master and Local columns indicate if the commits are local, or pushed to the master branch. The Version column indicates the version number given to the commit. The keyword column represents keyword in the commit message, if any are present.

and afterwards checks for keywords in the Git message and formats the new version name (lines 18–33). Jenkins is then triggered with a HTTP POST request containing the Jenkins job name, the version name of the release, and the commit id of the release commit as arguments. Finally, the library versions are updated in a global library object and the version name is printed to the user. The full code can be seen in Appendix D.1. The most recent versions of the different libraries are stored in a file together with the names of their repositories and Jenkins jobs.

9.7 Android Emulator Bug

When we started to work on improving build times, a new version of the Android Emulator Plugin (2.13) was released. This release contains a change in the way emulator startup is detected. It is necessary to detect when the Android emulator has started, so that APKs can be installed on it and tests can be run. Previously the plugin had checked whether the emulator has started up using `adb shell getprop dev.bootcomplete`. However, this property does not specify that the emulator has started fully, as it may still be in the boot animation when this property returns true. As such the check was switched to `adb shell getprop init.svc.bootanim` which indicates that the emulator is no longer in the boot animation, and thus have started fully. However, since the emulator in the plugin is started with the `-no-boot-anim` flag, the emulator has no boot animation. This is to improve boot time, but it also means that the new check for the boot animation will return true immediately, as there is no boot animation. This results in the plugin

```
def trigger_build(commit_msg, commit_id, libraries, repo_name,
    ↪ publish_snapshot=False):
    """
    Triggers a build on Jenkins.

    Parameters:
        commit_msg:       The contents of the commit message. Used for versioning.
        commit_id:        The id of the commit to build.
        libraries:        The known libraries and their versions.
        repo_name:        The name of the repository.
        publish_snapshot: If true, snapshots will be published.
    """
    version_name = ""
    for lib in libraries:
        if lib.name == repo_name and commit_msg != None:
            new_major = lib.major
            new_minor = lib.minor
            new_patch = lib.patch
            if "@major" in commit_msg.lower():
                new_major = str(int(lib.major) + 1)
                new_minor = "0"
                new_patch = "1"
                version_name = "%s.0.0" % (new_major,)
            elif "@minor" in commit_msg.lower():
                new_minor = str(int(lib.minor) + 1)
                version_name = "%s.%s.0" % (lib.major, new_minor)
                new_patch = "1"
            elif "@patch" in commit_msg.lower():
                version_name = "%s.%s.%s" % (lib.major, lib.minor, lib.patch)
                new_patch = str(int(lib.patch) + 1)
            else:
                if not publish_snapshot:
                    return
                version_name = "%s.%s.%s-SNAPSHOT" % (lib.major, lib.minor,
    ↪ lib.patch)
            # Send request
            try:
                send_build_request(lib.jobname, version_name, commit_id)
                # Apply lib changes (request succeeded)
                lib.major = new_major
                lib.minor = new_minor
                lib.patch = new_patch
                print_version_name(version_name)
            except (ConnectionError, Timeout) as e:
                print "ERROR TRIGGERING BUILD"
                print e
            return
```

Listing 9.4: Part of the Git hook responsible for setting library version names and triggering Jenkins (written in Python)

notifying that the emulator is ready too early, which can lead to potential errors when working with the emulator.

At first this change did not affect us, since building times when building submodules were sufficiently long for the emulator to have started fully regardless. However, as we improve building times with binary dependencies, we start getting errors because of the not yet started emulator. We submit a bug report for the plugin[3] and a pull request removing the `-no-boot-anim` flag for the compiler[4]. Removing the `-no-boot-anim` flag will result in slightly longer boot times for the emulator, but is necessary to detect that the emulator has started fully.

9.8 Evaluation of Build Times with Binary Dependencies

We measure the time of building the launcher after having implemented binary dependencies. To make a fair comparison, we run the launcher with the same Gradle tasks as previously measured (`check`, `connectedCheck`, `assembleRelease`), whereas in fact we now run every app with `clean`, `check`, `connectedCheck`, `increaseVersionCode`, `assembleRelease`, and `publishApkRelease`. The `increaseVersionCode` and `publishApkRelease` tasks make it possible to publish apps to Google Play. In addition, the `clean` task is run initially to ensure that a previous build does not affect the current build.

As mentioned in the previous section, we discovered a bug with the Android Emulator plugin in Jenkins after implementing binary dependencies that resulted in the very fast startups shown in Figure 9.1. We therefore compare the build times with the emulator starting correctly.

Figure 9.5 shows a stacked bar chart. *Old* shows the build times before the new dependency system with the emulator working correctly. *New (subset of tasks)* shows the build times with the build targets `check`, `connectedCheck`, `assembleRelease`. When running the same build tasks as before, there is an improvement in the build times of 43.6 %, as the dependencies Local-db, Giraf-component, and Oasis-lib are not built. Barcode-scanner is still built, as it is not a library used for any other projects than the launcher. The build time for Launcher has increased. This additional time is the time it takes to install the test APK on the emulator, which was previously performed during Giraf-component.

The complete build time with the added Gradle tasks (*New (all tasks)*) is, as expected, higher than when running fewer targets. The time for the Barcode-scanner and Launcher has increased because they run additional tasks. On comparable configurations, we achieve a significant time reduction of 43.6 % but some of these gains are eaten by the increased work added by the extra tasks and the resulting reduction in build time is just 13.7 %.

[3] https://issues.jenkins-ci.org/browse/JENKINS-27702
[4] https://github.com/jenkinsci/android-emulator-plugin/pull/49

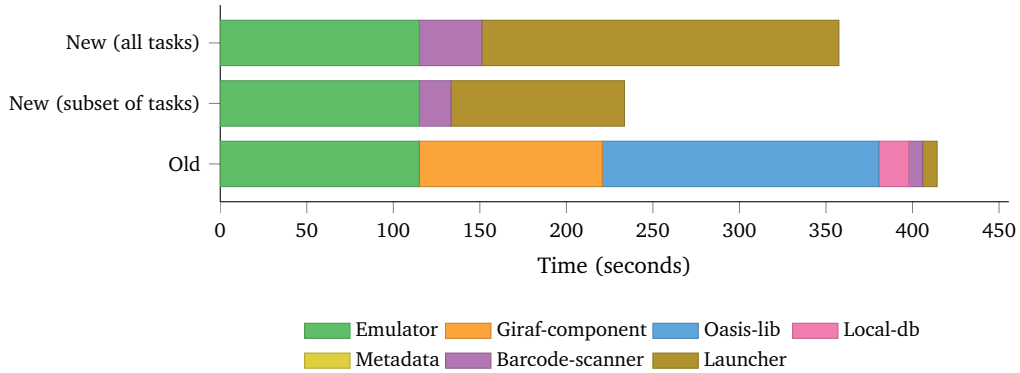


Figure 9.5: Comparison of Launcher timings

9.9 Non-Emulator Testing

As presented in the previous section, it takes a long time for the Android emulator to start. In addition, the emulator takes memory and CPU time from the actual build process, which is likely to make the build slower. We look at how to test Android applications and libraries sufficiently without using an emulator. Overall, we consider two ways of doing this: Testing on a physical device (connected by USB) and on the Java Virtual Machine (JVM).

9.9.1 Testing on a Physical Device

An alternative to the Android emulator is to run tests on a physical device. A physical device is typically faster than the emulator and it need not to boot every time we build. A downside of testing on a physical device is the expense of having one or more tablets only for testing. If we want to test on a wide range of Android devices, this can be a significant expense for a project without financial income. A single tablet may currently be sufficient for our project though, as we only test on a single emulator configuration anyway.

9.9.2 Testing on the Java Virtual Machine

Another option is to run tests on the Java Virtual Machine instead of the emulated Dalvik Virtual Machine. By not emulating the virtual machine, the test can be executed faster. However, this also means that the tests will not run on the Android operating system but on the operating system running the JVM. Because of this, some system features will have to be mocked, which in some cases may not provide a realistic simulation of the Android system. This may be acceptable, though, and we can eventually solve it by running delayed functional tests on an emulator at night when timing is no concern.

Robolectric

Robolectric [54] is a library for testing Android applications without an emulator or device. It contains implementations of the base Android SDK classes such that it can run on the Java Virtual Machine. It supports both unit testing and functional testing by providing access to UI elements and databases. To support this, however, the test case implementation differs slightly from those using the official Android test framework. Because of this, our existing tests need to be rewritten to support the library. In addition, it is not possible to run Robolectric tests on a device or emulator. As a result, it is necessary to have two tests, one using Robolectric and one using the Android test framework, if the test cases should run on both the JVM and an emulator or device.

While Robolectric is likely to speed-up the build times and seems to be relatively easy to use, the changes required to convert existing tests to Robolectric are too extensive for us to perform in this sprint. Therefore, we choose not to implement Robolectric. However, the library ought to be considered if build times need to be decreased further.

10 Sprint Review

We have solved most user stories this sprint, but monkey tests continues to be unfinished. This chapter evaluates the tasks we planned to do and the sprint as a whole on the multi-project level.

Chapter Organization This chapter is organized in the following fashion:

- In Section 10.1 we evaluate how the sprint went and whether we have solved our user stories;
- in Section 10.2 we evaluate how the sprint went on the multi-project level.

10.1 Sprint Goals

Set up Monkey Testing We did not manage to finish the setup of automatic monkey tests, due to unexpected technical difficulties. Therefore the monkey test user story remains in the product backlog.

Faster Build We had initially created some tasks aimed at reducing the time spent on testing, especially the emulator start up time. However, shortly after the sprint began, a patch was released to the Jenkins Android Emulator Plugin, which reduced the emulator start up time drastically. As such we down-prioritized the emulator related tasks. Later, we found out that it was a bug that was causing the build to start before the emulator finished booting. We have therefore ignored an important part of the build process in this sprint. This means that the build times increased significantly again after fixing the bug.

We have replaced submodule dependencies with binary files. This greatly improves the build time, as the dependencies do not have to be built every time a project is built. Overall we have improved the build times slightly. The Launcher app build time specifically has been improved by 13.7 %.

Uploading of Apps to Google Play We have successfully enabled automatic upload of GIRAF apps to Google Play. Every time an app successfully builds in Jenkins the app is automatically uploaded and released on Google Play on the alpha track. From there, the Google Play group can easily upgrade an alpha release to a beta or production release.

Test Case Installation Every night the newest build of each app is installed on an emulator to identify any installation problems.

Code Coverage Code coverage reports are generated for all GIRAF projects and the result are published in Jenkins. Detailed statistics are available on the job page,

and the percentage of lines of code covered for every project is visible on the job overview page.

We did not allocate enough time for writing the report this sprint. We had to catch up on an unfinished backlog with report items from sprint 1. We did estimate and schedule how much time was needed to complete the report for sprint 1. Our estimations proved accurate. However, we completely neglected to estimate and allocate any time for writing report about sprint 2, so we will be taking extra time in the next sprint for report writing in order to catch up.

10.2 Multi-Project Sprint Review

A common meeting among all groups is held to evaluate and reflect upon the process in this sprint.

It was noticed that chickens during the meetings of this sprint have been too noisy at times. We point out that chickens should find alternative means of communicating during meetings, and hope this will be enough. Otherwise, we will point it out during future meetings.

Also, some groups repeatedly did not show up to sprint meetings. We decide to find out the reasons behind it, knowing we cannot force them to show up. One group responds that they simply forgot the meetings, but will be more careful of remembering meetings in the future.

Two days between the GUI and the other two sprint planning meetings, suggested at the sprint 1 multi-project review, was introduced in this sprint. The feedback is positive and people are happy with this approach. As such we continue with it.

A suggestion is made allowing a product owner from DB and B&D to participate as pigs in the GUI sprint planning meeting. This seems like a good idea as they may have some useful input. We add this to the development method.

It is suggested by the semester coordinator to add a feature freeze. The suggestion does not come as a result of any concrete issues, but rather as a general suggestion. The suggestion is, however, quickly dismissed. Instead, we plead people to use common sense regarding implementing last minute changes before the end of a sprint. We also remind people that they should talk to affected groups before making a change and make daily integration.

Sprint 3

11 Sprint Planning

In this sprint, we focus on improving the development method and implementing monkey testing. We first attend the B&D sprint planning meeting, followed by an internal group sprint planning meeting.

Chapter Organization This chapter is organized in the following fashion:

- In Section 11.1 we describe the user stories we commit ourselves to during the B&D sprint planning meeting;
- in Section 11.2 we describe the sprint planning in our group and lists our tasks with reference to a user story.

11.1 B&D Sprint Planning

During the B&D sprint planning we choose to work on the following user stories in this sprint. They are labeled with a number in parentheses for reference. Notice that due to unfinished report business, we do not select large user stories. This allows us to catch up on the report writing.

Make Guidelines for Continuous Integration (1) We found out that some groups have a long-term branch that they only merge into master at the end of a sprint. Investigating this, people from other groups mentioned that they are not sure of how to do continuous integration. Because continuous integration is an important part of our development method, we will make some guidelines on how to use it.

Add a Guide on How to Do UI Test (2) Even though UI testing was implemented in sprint 1, no people from other groups were aware of the facility. We will make a small guide on how to do it.

Monkey Test (3) Last sprint we did not manage to setup monkey testing. The monkey tool does not support sending extra information to apps which is required to be able to start Giraf apps. In this sprint, we will make it possible to start apps without sending extra information when running monkey tests.

Specify the Scrum Process Used (4) Some groups have complained that the current documentation of the process used in the multi-project is too sparse. Therefore they have requested a detailed specification of the process.

Task	User Story	Estimation
Make guide to CI best practices	1	2
Make guide to UI testing	2	1
Support monkey testing in all apps	3	4
Investigate Scrum process with Group 3	4	1
Make process specification (+)	4	2
Missed tasks		
Setup subscription to monkey test reports in Jenkins	3	4
Original total		12
Total		14

Table 11.1: Sprint backlog for sprint 3, excluding report tasks. The tasks are listed in no particular order.

11.2 Group Sprint Planning

At our internal sprint planning we divide the chosen user stories into tasks and estimate them. For this sprint, we have a total of 55 half days of work. Table 11.1 shows the tasks we have committed to solve for this sprint. Tasks with a plus (+) are tasks that have been added during the sprint as they were discovered.

We make tasks for documenting the work from the previous sprint estimated for 25 half days. In addition, we estimate 11 half days for the report of sprint 3. The original total amount of work estimated for both the report and user stories is 51. This leaves 4 units of unspecified work. When we make the tasks for this sprint, we are unsure about the amount of work contained in user story 4. Thus we make the task about investigating the Scrum process with Group 1, which then result in the additional task *Make process specification*. During the sprint, we additionally add 8 half days of report tasks because we find more work. As such, we end up with a total of 59 half days.

12 Development Method Improvements

In this sprint, we work on specifying the development method for the multi-project, continuous integration and a guide on how to make Android UI tests.

Chapter Organization This chapter is organized in the following fashion:

- In Section 12.1 we review our development method, and make our specification available to all groups;
- in Section 12.2 we create a guide for the other groups on how to use continuous integration;
- in Section 12.3 we describe the creation of a guide to performing UI testing.

12.1 Specification of Development Method

While we have specified much of the development method in our report, there is currently only a very sparse specification available for the multi-project groups on the Redmine wiki. We therefore publish our specification in Chapter 2 (excluding Section 2.2) for the multi-project to consult.

In connection with specifying the development method, Group 1 approaches us with some feedback on the development method. They have made an analysis of some aspects of the multi-project where they feel the development method is not appropriate. The analysis can be found in Appendix K. The main points of concern are:

Dividing by Responsibility, Not Functionality The multi-project divides groups into three subprojects, each with a distinct responsibility. The analysis argues that this makes it hard to formulate user stories and display value to customers (Appendix K).

We agree that it is hard to formulate user stories. However we do not think this is caused by the multi-project structure, instead we believe it is difficult because it is not clear what a user story is. We discuss the problem with Group 1 and we agree that making a precise specification of how user stories are written will alleviate the difficulty concerning the formulation of user stories.

We have previously discussed the organization of the multi-project in Section 6.2, and at the time we decided that it was the best possible structure for our conditions. Now, as we have worked on the multi-project for a while, we agree with Group 1 that the subproject structure has some clear weaknesses, in particular as there

is a danger of gold-plating. In a subproject, all groups must have work to do, but the workload across subprojects is not the same and it changes throughout the multi-project. This means that some groups may work on unimportant things (from the viewpoint of the success of the multi-project). We therefore suggest not to divide subprojects this way in future semesters, but rather use a single backlog from which all groups can select any item. This ensures that work can be selected in a prioritized way. We expect that the customer will, to a higher degree than today, get what they need the most. We are aware that this may decrease the overall productivity (in terms of functionality over time), as groups do not necessarily specialize in a single technology. However, because groups commit themselves to solve certain backlog items (in contrast to being assigned to solve them), we expect that groups generally tend to work on backlog items that they feel comfortable with. We accept a small decrease in the overall functionality/time ratio if it results in giving the costumers a usable product fast.

No Real Product Owner There seems to be a misunderstanding of the product owners of the multi-project: In Group 1's analysis, they mention a top level product owner (being the external customers), yet there is no such role in the multi-project. They also argue that the subproject product owners (POs) do not sufficiently understand the requirements and mention that there should be a single person being the product owner to avoid conflicting prioritization (Appendix K).

We do, however, not believe that it would be possible for a single person to be PO, as this person would then be fully occupied with this, which is not desirable in an educational context. Having spoken to the product owners we think that they do actually understand the items in the backlog. The confusion is likely due to a misunderstanding of the development method that should be remedied by having a precise user story specification available.

The Product Is Seeing Very Limited Use The analysis states that despite the GIRAF apps have been in development since 2011, there is no major use of the system (Appendix K). While this is indeed correct, it is a mistake to believe that a sprint in Scrum should result in a releasable product. It might take many sprints before this can be done [39]. Group 1 suggests defining a *minimum viable product* (MVP) that has exactly the features that allows the product to be deployed. We discuss this suggestion with Group 1 and agree that this will help focus on the most important parts of the multi-project. We add the backlog item of defining a MVP to the backlog, so that it may be worked on in the next sprint.

Only Features and Bugs in the Product Backlog The analysis mentions that user stories used in the backlog only described features and bugs, making it difficult to formulate, prioritize, and work on technical work, such as major refactoring, and knowledge acquisition, such as investigating what library to use to solve a certain problem (Appendix K).

We discuss this concern with Group 1 and agree to add new types of items to the

backlog. A result of having these different types in the backlog is that we make it clear for multi-project members that it is acceptable to spend time on knowledge acquisition and technical work, not only features and bugs.

To summarize, together with Group 1, we decide to make the following suggestions for next semester:

Divide by Functionality The multi-project should not be structured by subprojects as this semester, but rather every group should be able to pick any item of a common backlog. They should still use the hierarchical Scrum of Scrums structure to lessen workload on product owners. We suggest that they form new "dynamic" subprojects every sprint. After each group have selected their product backlog items for their release backlog they should be divided into a fitting number of subprojects. All groups in each subproject should work on related parts of the multi-project, e.g. if Group A and Group B are both working on product backlog Items which request changes in the same app, they should be part of the same subproject during that sprint. Each group must be in exactly one subproject. This approach requires greater discipline regarding continuous integration, as it is expected that groups will work closer together, as it is likely that more than one group will work on an app at a given time. In addition, it requires that groups are realistic about which backlog items they commit themselves to solve.

Work on a Minimum Viable Product Create an MVP to get the system in use quickly and satisfy the customer and thus get valuable feedback from the users.

We make the following changes to the development method for the final sprint in this semester:

A Single Backlog Currently, the backlog is separately managed by the subproject POs in separate lists. To make it easy to work with the backlog a single backlog should be made available. This is also necessary for the next semester to easily move onto a structure of functionality division.

New Backlog Items We add technical work and knowledge acquisition items to the backlog to make it easy to formulate and work on those items. We specify a template for formulating user stories, and add *conditions of satisfaction* for user stories. We also add *constraints* to the backlog that specify constraints for one or more backlog items.

12.2 Specification of Continuous Integration

We have committed ourselves to solve a user story proposed by the developers, which states that they want a guide about continuous integration. We observe that many developers *want* continuous integration (it was a high priority user story in sprint 1), but do not really understand how to use it. In the sprints up to now, we have observed

developers working on features on isolated branches for several weeks without integrating it with the master branch. Likewise, only a single group has written automated tests for their code. This is a big concern, because agile development and continuous integration highly depends on automated testing. In sprint 1, we wrote a sample test for each GIRAF project to make it easy for developers to start writing tests. In sprint 2, we added a code coverage measure to each GIRAF project. We find, however, that this has not resulted in more tests written. Therefore, solely writing how to do does not seem to be sufficient. For this continuous integration guide, we try to argue more about why to do as we propose in order to convince developers that they should indeed follow the guidelines.

The CI guide has been released on the Redmine wiki, and is included in Appendix G. Instructions on when to release to Artifactory has been added to the Redmine guide on dependencies. The instructions appears as specified in Section 9.4.1.

12.3 Guide on Making Android UI Tests

To create awareness of the opportunity for groups to have UI tests in the apps, we write a short guide on how to do this, as requested by the multi-project groups. This includes both running UI tests locally as well as in Jenkins upon each build. We do not want to spend much time on doing this, since a great amount of material is already available on the Internet. Therefore, we only find the appropriate material as well as comment on special considerations for the GIRAF projects. These special considerations include how to setup the database with dummy data, and faking login credentials. The Android UI test guide can be seen in Appendix H.

13 Making Monkey Testing Work

We have tried implementing monkey testing in the two previous sprints. In this sprint, we continue with this and finally succeed in setting up monkey testing for all GIRAF apps.

Chapter Organization This chapter is organized in the following fashion:

- In Section 13.1 we make a simple app that inserts dummy data into the local database;
- in Section 13.2 we modify all GIRAF apps so that they accept a monkey test and use the dummy data inserted into the database.

13.1 Dummy Database Inserter

To avoid downloading the full database before starting the monkey test, we create a new Android application which inserts dummy data into the database. The application is very simple and the main part is shown in Listing 13.1. It starts a thread in line 6 which inserts the data using a method provided by the database library (line 18). When inserted, the application closes. We detect when the applications has closed to know when the data has been inserted. In line 26, we call `System.exit(0)` which ensures that the application is completely closed (killed), as it otherwise would just be in an idle state.

Having installed the dummy database inserter app onto a device, it must be started before running any monkey test. It is vital that the monkey test does not start before the dummy data has been inserted. Otherwise the database will not be populated with usable data.

We therefore make a script, seen in Listing 13.2, that starts the dummy database inserter app and waits for it to finish by busy waiting. It does so by first starting the app in line 2. It then runs a while loop (lines 6–10) that uses `adb shell ps` to list all processes running on the Android device, and then piping that to `grep` to check for the specific app. The loop runs for as long as the output is not empty.

Busy waiting is generally to be avoided, as it wastes CPU time [60, ch.2]. An alternative would be to block the process and have the tablet notify the process that it has finished. However, this is time consuming to implement. To avoid too much waste of time, we insert a sleep of two seconds, thus blocking the process and freeing up CPU time.

13.2 Adapting GIRAF Apps for Monkey Testing

Each GIRAF application expects being started from the Launcher, so that information about the current user is passed to the app. This prevents us from running monkey tests

```java
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        new DataInserter(this).execute();
    }

    private static class DataInserter extends AsyncTask<Void, Void, Void> {
        private Activity context;

        public DataInserter(Activity context) {
            this.context = context;
        }

        @Override
        protected Void doInBackground(Void... params) {
            new Helper(context).CreateDummyData();
            return null;
        }

        @Override
        protected void onPostExecute(Void aVoid) {
            // Force exit app (so we can detect it)
            context.finish();
            System.exit(0);
        }
    }
}
```

Listing 13.1: Dummy database inserter `MainActivity` (written in Java)

```bash
#!/bin/bash
/srv/android-sdk-linux//platform-tools/adb shell am start -n
    ↪ dk.aau.cs.giraf.dummydbinserter/dk.aau.cs.giraf.dummydbinserter.MainActivity

IS_RUNNING="$(/srv/android-sdk-linux//platform-tools/adb shell ps | grep
    ↪ dk.aau.cs.giraf.dummydbinserter)"

while [ "$IS_RUNNING" ]
do
    IS_RUNNING="$(/srv/android-sdk-linux//platform-tools/adb shell ps | grep
    ↪ dk.aau.cs.giraf.dummydbinserter)"
    sleep 2
done

echo "finished dummy insertion"
```

Listing 13.2: Bash script which starts and waits for dummy database inserter

```
1  public class MainActivity extends FragmentActivity {
2    // ...
3
4    protected void onCreate(Bundle savedInstanceState) {
5      // ...
6      Bundle extras = getIntent().getExtras();
7      getProfileFromExtras(extras);
8      checkIfValidProfile();
9      // ...
10   }
11
12   private void getProfileFromExtras(Bundle extras) {
13     // ...
14     } else if (extras.containsKey(EXTRAS_PROFILE_CURRENT_GUARDIAN_ID)) {
15       signInWithGuardianId(extras.getInt(EXTRAS_PROFILE_CURRENT_GUARDIAN_ID));
16     } // ...
17   }
18
19   // ...
20 }
```

Listing 13.3: Original User Manager login procedure (written in Java)

on the apps, as the monkey test will start the app without providing this information, resulting in crashes or failure to start. We circumvent this by detecting whether an app is started by a monkey test; if so, we use a test user received from the database to login. The process varies for each app. We show how to implement the change for the User Manager app in the activity `MainActivity`.

The original code can be seen in Listing 13.3. When `MainActivity` is created, it gets the login information provided by the Launcher (line 6). This information is then passed onto `getProfileFromExtras()` that gets a profile from this information. Line 12–17 shows a snippet of this method. It calls `signInWithGuardianId()` with the ID of a guardian.

In the modified version of *user manager* that supports monkey testing, seen in Listing 13.4, we add an if-statement (line 7) that checks if the app is started by a monkey test. If it is, we make a new `Helper` (line 8), which is a database helper that enables us to get all guardians and select the ID of the first guardian (line 10). Instead of calling `getProfileFromExtras()`, we call `signInWithGuardianId()` directly. If the app was not started by a monkey test we proceed as normal (lines 12–15).

In this manner, we circumvent the expectations of the apps when running monkey tests. We create jobs in Jenkins which run monkey tests on all apps every night.

We did not have time to set up automatic notifications to developers whenever a monkey test fails. While this is unfortunate, it was not part of the monkey test user story. In any case it is an important feature to have so that developers are informed of errors in apps automatically, rather then seeking the information themselves. We add this as a user story to the product backlog. However, they can manually subscribe to an RSS feed of the test status.

```java
public class MainActivity extends FragmentActivity {
  // ...

  protected void onCreate(Bundle savedInstanceState) {
    // ...

    if (ActivityManager.isUserAMonkey()) {
      Helper h = new Helper(this);

      signInWithGuardianId(h.profilesHelper.getGuardians().get(0).getId());
    }
    else {
      Bundle extras = getIntent().getExtras();
      getProfileFromExtras(extras);
    }

    checkIfValidProfile();

    // ...
    }

  // ...
}
```

Listing 13.4: Updated User manager login procedure for monkey testing (written in Java)

14 Sprint Review

We have succeeded in accomplishing all user stories this sprint but missed a non-crucial task. In this chapter, we evaluate the tasks we planned to do and the sprint as a whole on a multi-project level.

Chapter Organization This chapter is organized in the following fashion:

- In Section 14.1 we evaluate how the sprint went and whether we reached our goals on a group level;
- in Section 14.2 we evaluate how the sprint went on the multi-project level.

14.1 Sprint Goals

Make Guidelines for Continuous Integration We have fulfilled the desire of the developers for a clear central specification of the process. The specification is available on the Redmine Wiki, and resembles Chapter 2. The specification is necessary as it is time consuming to explain it over and over, and because the process sometimes changes. A central wiki entry helps the developers keep up to date.

Add a Guide on How to Do UI Test We have specified how to set up and run UI testing in a guide which is available on the Redmine Wiki. The guide links to the testing guide [31] of the Android Developer website, as it is clear and comprehensive. Our guide just adds details specific to the multi-project, e.g. how to initialize the database with dummy data.

Monkey Test In this sprint, we have enabled monkey testing for all apps. We have created a new app for inserting dummy data into the local database. This app is run before the app that must be monkey tested. In that way, apps can be run without downloading the real database, which can take 10 minutes or more. There already existed jobs in Jenkins for monkey testing, and these have been updated to use the dummy data. We did not manage to fulfill the task of setting up a subscription based notification system for developers, as we discovered report related tasks that we prioritized more highly than this task. In any case this task is actually not necessary to fulfill the user story — it should not have been created in the first place.

Specify the Scrum Process Used Group 1 approached us last sprint with an analysis of the development method. The analysis raised a number of critiques, which we discussed with Group 1. From this discussion we, in collaboration with Group 1, revised the development method. We specified how user stories are formulated and added new types of product backlog items.

14.2 Multi-Project Sprint Review

Up to the end of the sprint there were some issues regarding the Launcher app. An update of the database library had rendered the Launcher unusable. This meant increased workload at the end of the sprint to fix the issue. At the multi-project sprint review it is explained that the error had been there for a while, but was only discovered at the end of the sprint, as the GUI groups did not always use the newest version of each app. In addition, the DB PO mentions that other bugs were introduced because of a large merge with the master branch.

We reiterate the importance of performing daily integration with the master branch, to ensure that no major merges are performed that can introduce numerous errors. The semester coordinator also note the importance of GUI groups always using the latest versions of all apps to catch such errors as quickly as possible.

Sprint 4

15 Sprint Planning

This sprint planning proceeds as specified by the development method: We first have a sprint planning in the subproject followed by sprint planning in the group.

Chapter Organization This chapter is organized in the following fashion:

- In Section 15.1 we describe the backlog items we commit ourselves to during the B&D sprint planning meeting;
- in Section 15.2 we describe the sprint planning in our group and list our tasks with reference to a backlog item.

15.1 B&D Sprint Planning

In the previous sprint we specified the backlog items used in the multi-project and how to formulate user stories (Section 12.1). This specification is followed this sprint.

During the B&D sprint planning we choose to work on four user stories and one technical work item. They are labeled with a number in parentheses for reference.

Each user story now has a number of *conditions of satisfaction*. These conditions of satisfaction are listed as bullets below each user story. The conditions have been elicited by asking our customers and our product owner.

(1) As a developer I want libraries to have higher priority than other jobs in the build scheduling in Jenkins so that libraries are not delayed when several people are reliant on them.

- Jobs must not be starved
- Libraries must be prioritized higher than other job types

(2) As a developer I want monkey tests to run on the debug version of apps so that they can be tested on a test database.

- Monkey tests must run on debug APKs

(3) As a developer I want an easy way to download and install all apps so that it is easy to test the apps combined, and easy to show them to the external customers.

- Users should not have to install additional software onto their computers
- Users should be able to merely run a program that automatically downloads and installs all apps on a device
- After running the program, the device must contain the newest version of all apps

(4) As a future developer I want to know what is in Jenkins and how it is structured so that it is easy to start working with it

- Document how to configure Jenkins jobs
- Document configuration of authentication
- Document files used for automation

(5) As a developer I want to have a screenshot taken when a monkey test fails so that I can get feedback about the failure

- The last screen at a crash must be saved

The following is the technical work item we work on in sprint 4. This technical work item is not formulated as a user story, as it is hard to have conditions of satisfaction for it. The build time can almost always be decreased, and so it would be a never-ending user story. We choose to work on it in this sprint, as it was prioritized highly by the other groups.

(6) Decrease Job Build Times in Jenkins When the queue in Jenkins is long, jobs can take a very long time to build. As we discussed in Section 9.9, the build times can be further reduced by working on the emulator part.

15.2 Group Sprint Planning

At our internal sprint planning we divide the chosen backlog items into tasks and estimate them. For this sprint, we have a total of 70 half days of work. Table 15.1 shows the tasks we have committed to solve for this sprint. Tasks with a plus (+) are tasks that have been added during the sprint as they were discovered. Tasks with an estimation of 0 have been estimated as such, because they take virtually no time. There are three tasks in the sprint backlog which are not directly related to any backlog item (marked by n/a as their backlog item). We do these tasks as some acute needs developed during the sprint.

In addition to the tasks related to solving our chosen backlog items, we also work on report related tasks to finish our report, such as introduction, conclusion, etc. The total amount of time estimated for report tasks during this sprint is 56.

The total estimate of 18 for the original tasks is quite low. This is because we do a number of spikes, as indicated in parentheses for those tasks. We do these spikes because we have many uncertainties during this sprint. The final total estimate is therefore 37. The estimation for all tasks (report and non-report tasks) is 93. This far exceeds the time we have available for this sprint. Since we have a week following the sprint to do the final report changes, we postpone some report tasks.

Task	Backlog Item	Estimation
Investigate conditions of satisfaction for our back log items	n/a	2
Make pre-commit hook (+)	n/a	1
Check monkey tests after server crash (+)	n/a	2
Install Jenkins priority plugin	1	1
Choose schedule method for priority plugin	1	2
Give metadata high priority (+)	1	0
Make monkey tests use debug APKs	2	1
Place debug APKs in a specific directory	2	1
Make script for downloading and installing newest APKs	3	2
Identify areas for Jenkins structure documentation (spike)	4	2
Write about Jenkins structure	4	2
Write about Jenkins files	4	2
Make monkey tests take screenshot and publish them	5	0
Make ADB-wifi app (+)	6	2
Setup Jenkins Job for Simiasque (+)	6	1
Investigate usage of physical tablets (spike)	6	1
Run monkey test without emulator plugin (+)	6	2
How do we automatically connect to tablets? (spike)	6	2
Connect to all devices wirelessly from server (+)	6	2
Uninstall APKs on all devices (+)	6	1
Setup router (+)	6	2
Only start an emulator when there are no connected devices (+)	6	6
Original total		18
Total		37

Table 15.1: Sprint backlog for sprint 4, excluding report tasks. The tasks are listed in no particular order.

16 Further Improving Build Times

This chapter describes our work on the technical work item *Decrease Job Build Times in Jenkins*. The developers want more rapid response to builds failing or succeeding. As explained in Section 9.8, the emulator start-up time is a significant part of the build time, taking approximately two minutes. The emulator is used for running the tests, and as such we have to find an alternative method of running the tests.

Chapter Organization This chapter is organized in the following fashion:

- In Section 16.1 we choose to use non-emulator testing to decrease the emulator time usage during building, and we describe our plan for implementing this;
- in Section 16.2 we setup a pool of tablets to be tested and we describe the scripts that perform this task;
- in Section 16.3 we enable testing on the tablets, and create an app to the tablets that automates this process as much as possible;
- in Section 16.4 we describe the configuration of Jenkins such that it only starts an emulator if there are no connected devices;
- in Section 16.5 we present the updated flow for jobs in Jenkins in order to successfully run tests on tablets. This includes running scripts to connect to devices, uninstall old APKs, build and install the necessary APKs, and disconnect devices;
- in Section 16.6 we evaluate the build times of the Launcher project with an emulator and with a tablet.

16.1 Selecting a Non-Emulator Test Method

As explained in Section 9.9 there are two ways of running tests without an emulator: testing on a physical device, and testing on the Java Virtual Machine. Testing on a physical device is the only way to realistically run tests. All groups have a tablet laying unused much of the day as well as all night, and we will utilize these unused tablets. This way, we can test on multiple different devices and Android versions without significantly increasing the build times and resource use on the server, which would be the case when testing on emulators. However, it will be too tedious connecting these tablets physically to the server (the server is behind locks in the basement of Aalborg University) and we most likely will not get permission to physically connect USB devices to it.

We want to create a pool of available tablets, such that groups in an easy way can provide their tablets for testing. Our plan is to setup a wireless router in the vicinity of our group rooms. The wireless router broadcasts a Wi-Fi signal, and any tablet connected

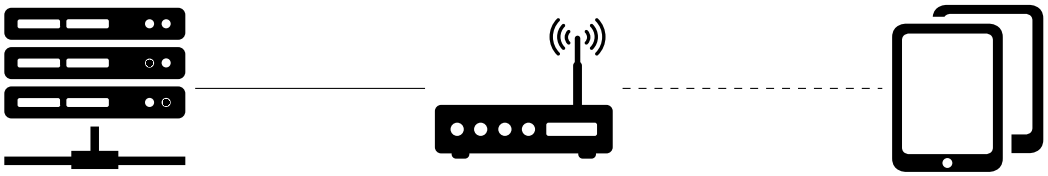


Figure 16.1: Non-emulator testing setup. The line from the server is a cabled LAN connection to our wireless router. The dashed line from the wireless router is the wireless connection to each tablet.

to this Wi-Fi will automatically be in a pool of connected tablets, potentially available for testing. The server is able to communicate with the wireless router through the network at Aalborg University. An illustration of the setup can be seen in Figure 16.1.

16.2 Creating a Pool of Test Tablets

In order to transform a common wireless router into a device that maintains a pool of tablets, we flash it with a custom firmware. A custom firmware allows us to access the device as root user and reprogram parts of it. We tried flashing with the *DD-WRT* [17] firmware, an open source router firmware. However, we found no build of the firmware that worked properly on our device. Either SSH or some crucial programs did not work (also documented in the DD-WRT forums by other users), or the build was known to brick the router. We therefore choose to flash with *OpenWRT* [47], another open source router firmware. We choose OpenWRT because it is compatible with our device and have scripting support. After flashing with OpenWRT, we setup the basic system settings and configure the Wi-Fi network as one normally will in any home router.

All tablets connected need static IP addresses as well as port forwarding to them. Therefore, we also need to partially disable the DHCP server on the router. A DHCP server automatically assigns any connecting client with a temporary IP address. We disable this, which means that any connecting client will not get an IP address.

Now, we create a script which searches for any connected clients without an assigned IP address, and then assigns IP addresses to them, as seen in Listing 16.1. It does this by retrieving a list of connected MAC addresses (line 4). `iwinfo wlan0 assoclist` outputs a list of devices connected to the wireless interface `wlan0` of the router. An example of this list can be seen here:

```
1 60:36:DD:06:6F:9E  -46 dBm / -95 dBm (SNR 49)  0 ms ago
2         RX: 144.4 MBit/s, MCS 15, 20MHz, short GI        966 Pkts.
3         TX: 72.2 MBit/s, MCS 7, 20MHz, short GI          734 Pkts.
4
5 1C:66:AA:5B:56:C2  -27 dBm / -95 dBm (SNR 68)  1240 ms ago
6         RX: 65.0 MBit/s, MCS 7, 20MHz                   1902 Pkts.
7         TX: 58.5 MBit/s, MCS 6, 20MHz                    609 Pkts.
8
9 // ...
```

```
1  #!/bin/sh
2
3  # Retrieve mac addresses of connected devices on wlan0:
4  maclist=$(iwinfo wlan0 assoclist | grep -E -o '[[:xdigit:]]{2}(:[[:xdigit:]]{2}){5}'
       ↪ | awk '{print tolower($0)}')
5
6  . /lib/functions.sh
7  config_cb() {
8    config_type="$1"
9    if [ "$config_type" = "host" ]
10   then
11     option_cb() {
12       option_name="$1"
13       value="$2"
14       if [ "$option_name" = "mac" ]
15       then
16         knownmacs="$knownmacs $value"
17       fi
18     }
19   fi
20 }
21
22 config_load dhcp
23
24 for x in $maclist
25 do
26   echo $knownmacs | grep "$x" > /dev/null
27   if [ "$?" != "0" ]
28   then
29     # We found a new device. Call this script to setup static IP and port forward:
30     /bin/setup_staticip_portforward.sh $x
31   fi
32 done
```

Listing 16.1: Shell script that discovers new devices and assigns a static IP address and port forward by calling the script below

We pipe this to `grep` which extracts the MAC addresses. Following we pipe them to `awk` which makes the MAC addresses lowercase.

At lines 6–22 we generate a list of MAC addresses with an assigned IP address. First, the `functions.sh` library is imported, and we define a callback function `config_cb()`, which is called by `config_load` in line 22 every time a configuration line is parsed. In the callback we first check if the configuration type is `host`, and then a new callback function `option_cb()` is defined, which is called for every option of the configuration type `host`. If the option name is `mac` we append it to a list of the MAC addresses.

We now have a list of the MAC addresses of connected devices and a list of MAC addresses with an assigned static IP address. We cross-reference these two lists, and all MAC addresses without an assigned IP address will get one as well as a port forward rule in the firewall (line 30). Since we control the IP addresses and port forwards given, we

can safely just assign an incremented IP address and port number. The script called in line 30 can be seen in Listing 16.2. This script calls the `uci` tool, Unified Configuration Interface, which is a way of changing configuration in the OpenWRT system.

A new device will thus trigger the following changes in the router:

1. We save an entry in the DHCP settings, such that the MAC address of the new device will be linked to the next free IP address (incremented)

2. We save a port forward entry in the firewall settings, such that any request to a port (e.g. `9001`) will be assigned the IP address of a device and port `5555`.

We have the option to run this script either in a loop, busy-waiting most of the time, or to run the script periodically in a cron job. If the script crashes, a cron job will ensure that it runs again. However, the minumum interval of a cron job is one minute. We find waiting up to a minute reasonable — it is only the first time connecting. After this first time, the devices will connect and receive their static IP addresses instantly. Because of the stability advantage, we choose to run the script in a cron job every one minute.

Now that tablets can connect to the router in a controlled way, we want to enable Jenkins to request a list of the port numbers of the connected tablets. OpenWRT already runs a small HTTP server for its configuration GUI. We create a new script, seen in Listing 16.3, that returns a list of the port numbers of all connected devices. This script is executed when a HTTP GET is requested to `http://⟨router-ip⟩/cgi-bin/devices`. We have enabled the HTTP server to be reached from outside the router in the router firewall. The script works by retrieving a list of connected MAC addresses (line 8), cross-referencing this list with a list of the current DHCP leases, which returns the IP addresses of the connected devices (lines 12–15). This list of IP addresses is cross-referenced with the firewall configuration, where we search for a port forward for each IP address (lines 18–40). The ports are returned (line 42). The configuration of the router can bee seen in Appendix J.

16.2.1 Creating Route from Server to Tablet

Testing the above setup reveals that the server and our router is separated on the network. The only traffic we can get through is ping. We investigated the cause of this with the help of IT Services at Aalborg University, and they moved our router to the same network as the server (determined by its MAC-address). We can now access the router from the server, which means we can access the tablets from the server (because of port forwarding on the router).

16.3 Wireless ADB App

The Android Debug Bridge[1] (ADB) tool supports device communication over Wi-Fi out-of-the-box [29]. This is very convenient for us, because the interface for communication

[1] Android SDK tool which manages communication between a computer and a device.

```
#!/bin/sh

# ...

# Allocate next ip and port
NEXT_IP_EXT="$(cat /usr/next_ip)"
NEXT_PORT="$(cat /usr/next_port)"
DEST_IP="192.168.1.$NEXT_IP_EXT"
SRC_PORT="$NEXT_PORT"
# Update ip/port files
echo "$((NEXT_IP_EXT+1))" > /usr/next_ip
echo "$((NEXT_PORT+1))" > /usr/next_port

uci batch <<EOF
add dhcp host
set dhcp.@host[-1].mac=$1
set dhcp.@host[-1].ip=$DEST_IP
commit dhcp
add firewall redirect
set firewall.@redirect[-1].target=DNAT
set firewall.@redirect[-1].src=wan
set firewall.@redirect[-1].dest=lan
set firewall.@redirect[-1].proto='tcp udp'
set firewall.@redirect[-1].src_dport=$SRC_PORT
set firewall.@redirect[-1].dest_ip=$DEST_IP
set firewall.@redirect[-1].dest_port=5555
commit firewall
EOF

uci commit dhcp
uci commit firewall

logger "New device $1 added with IP $DEST_IP and port $SRC_PORT"

/etc/init.d/dnsmasq reload
```

Listing 16.2: Shell script that sets up static IP address and a port forward for a given MAC address

```
#!/bin/sh
echo "Content-type: text/plain"
echo # Must be here.

# ...

# Retrieve mac addresses of connected devices on wlan0:
maclist=$(iwinfo wlan0 assoclist | grep -E -o '[[:xdigit:]]{2}(:[[:xdigit:]]{2}){5}'
    ↪ | awk '{print tolower($0)}')

# Get the IP for each connected mac adddress:
regex_ip="\([0-9]\{1,3\}\.\)\{3\}[0-9]\{1,3\}"
for x in $maclist
do
  iplist="$iplist $(sed -n -e "s/^\(.*$x\) \($regex_ip\) \(.*\)/\2/p"
    ↪ /tmp/dhcp.leases)"
done

# Find matching port number from firewall redirection configuration:
. /lib/functions.sh
config_cb() {
  type="$1"
  config_name="$2"
  if [ "$type" = "redirect" ]
  then
    option_cb() {
      option_name="$1"
      value="$2"
      for y in $iplist
      do
        if [ "$option_name" = "dest_ip" ] && [ "$value" = "$y" ]
        then
          local resultvar
          config_get resultvar "$config_name" "src_dport"
          DEBUG echo "$config_name, $resultvar"
          portlist="$portlist $resultvar"
        fi
      done
    }
  fi
}
config_load firewall

echo $portlist | tr " " "\n"
```

Listing 16.3: Shell script that returns port numbers of connected devices

between computer and device is the same no matter if the devices are connected by wire or not. To connect a device wirelessly to a computer, the following steps are to be performed:

1. Connect the device to the computer using USB
2. Run `adb tcpip ⟨port⟩` to enable Wi-Fi debugging
3. Disconnect the devices
4. Run `adb connect ⟨ip⟩:⟨port⟩` to connect to the device with the specified ip and port

These steps must be performed each time a device is rebooted. We want to automate this process, and we can do that by setting at specific property in an Android configuration file [14]. Wireless ADB is enabled by executing the following commands in the Android shell:

```
1 su
2 setprop service.adb.tcp.port ⟨port⟩
3 stop adbd
4 start adbd
```

To disable it again, the port property is simply set to `-1` instead of the port. However, doing this requires root permission on the device. We therefore require devices to be rooted in order to be part of the tablet pool. The expected port in the ADB tool is 5555, so we use that on the device.

To make it easy for developers to enable and disable wireless ADB (and by that add a device to the tablet pool), we develop an Android app to do this, shown in Figure 16.2. By making it easy to join the tablet pool, we expect that developers are likely to use this feature when they do not use the device.

The app does, however, not support safe disconnection from the testing pool. If developers disconnect their devices, they may do it during a build. The build will then result in failure due to premature disconnection. We add this as a user story to the product backlog to find and implement a solution to this issue.

16.3.1 Android App Implementation

To run the shell script in the app, we execute the code shown in Listing 16.4. When we execute `su` (line 5), the device shows a dialog which asks the user for root permission. After accepting, every subsequent `su` call will be executed without need for permission. The method returns the exit code of the command so it can be handled accordingly by the caller. If the device for some reason is rebooted, we make sure to automatically enable wireless ADB when the device is started. We do this by declaring a `BroadcastReceiver`, which is automatically triggered when the device is booted. The receiver is declared in the Android manifest file and implemented as shown in Listing 16.5. The `onReceive()` method (lines 3–8) is called when the broadcast is received. We set the state of wireless ADB according to the user's preference (lines 20–30). We also set a notification (lines 22

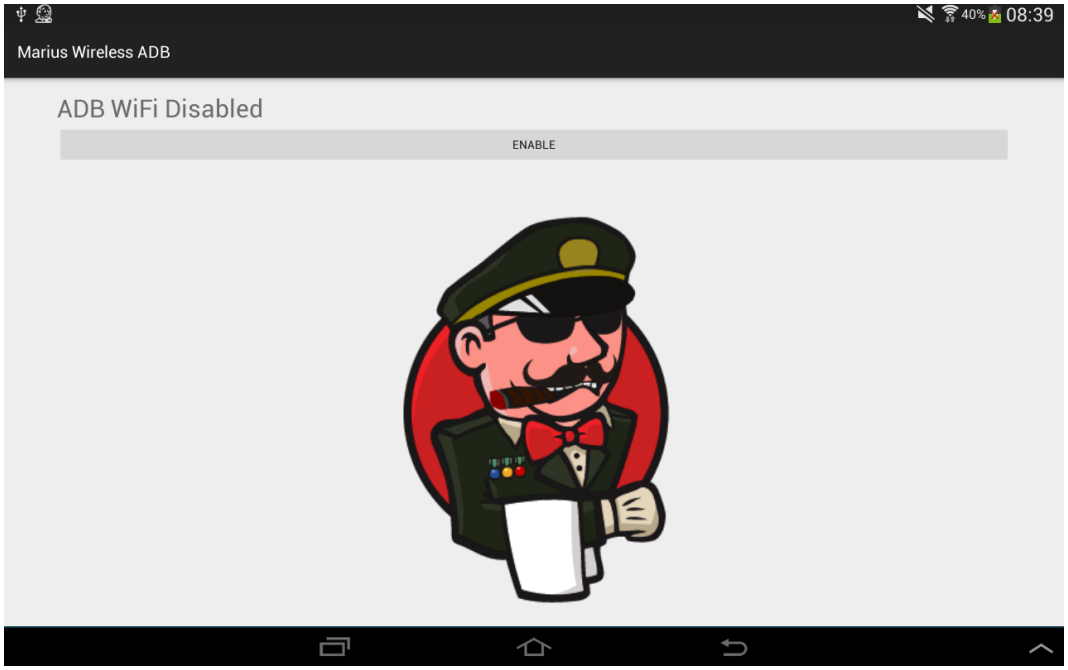


Figure 16.2: Wireless ADB app

```
public static int enableWifiAdb() throws IOException, InterruptedException {
  Process process = null;
  DataOutputStream os = null;
  try {
    process = Runtime.getRuntime().exec("su");
    os = new DataOutputStream(process.getOutputStream());
    os.writeBytes(String.format("setprop service.adb.tcp.port %d\n", ADB_TCP_PORT));
    os.writeBytes("stop adbd\n");
    os.writeBytes("start adbd\n");
    os.writeBytes("exit\n");
    os.flush();
    int exitValue = process.waitFor();
    process.destroy();
    return exitValu
  } finally {
    if(os != null) {
      os.close();
    }
  }
}
```

Listing 16.4: Enable wireless ADB in Android (written in Java)

and 27) to show the ADB state to the user, and start a background service which keeps the screen and Wi-Fi on while testing (lines 23 and 28).

16.4 Adapting Jenkins to Use Physical Devices

We currently use the Jenkins Android Emulator Plugin to run an emulator during each build. However, when one or more tablets are available for testing, we do not want to start an emulator. The plugin cannot be configured to start an emulator only if there are no physical devices attached. We decide to modify the Android Emulator Plugin to perform a check before starting an emulator. We modify the plugin such that it provides an option to check whether there are devices available, and then only to start an emulator if there are no devices already connected. We have submitted a pull request with the changes[2]. The maintainer of the plugin can choose to merge the request into the master branch of the plugin. This would be an advantage for us, as we the do not have to maintain our fork of the plugin.

Listing 16.6 contains the main addition to the plugin: The method which checks for connected devices. The main part of the method is preparing for the call in line 14, and parsing the output afterwards. The call is to a method which sends a command to a tool provided by the Android SDK. The call corresponds to writing `adb -d devices` in the terminal, which lists all connected devices. The parameters, in order, to the call are:

- A `Launcher`, which is an abstraction of an process call, which blocks the calling thread until the operation is completed
- An environment of variables which the process is run in. The most important variable is the `ANDROID_ADB_SERVER_PORT` variable, as it decides which port ADB uses.
- An `OutputStream` for standard output. This provides the output of the command
- An `OutputStream` for the error output of the command
- An `AndroidSDK` which contains information about and methods related to the installed Android SDK
- The tool to run. An Enum with information about the tools provided by the SDK
- The actual command to send to the tool, formatted as a string. In this case it is `-d devices`.
- The `FilePath` to the Jenkins workspace if relevant. In this case it is not, so we just send a `null` value.

After the call, the standard output is converted to a string, and the method `getDeviceNames()` converts the string into a list of device names. If the list is populated, then there

[2] https://github.com/jenkinsci/android-emulator-plugin/pull/50

```java
public class BootReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    if(intent.getAction().equals(Intent.ACTION_BOOT_COMPLETED)) {
      Log.i("giraf", "Starts adb after boot");
      enableAdb(context);
    }
  }

  /**
  * Enables adb if the preference is set.
  * @param context The application context.
  */
  private void enableAdb(Context context) {
    SharedPreferences prefs = PreferenceManager.getDefaultSharedPreferences(context);
    boolean enabled = prefs.getBoolean("adb_enabled", false);
    // Enable/disable adb wifi
    try {
      if (enabled) {
        if(AdbUtils.enableWifiAdb() == 0) {
          AdbUtils.showNotification(context, true);
          AdbUtils.startWakelockService(context, true);
        }
      } else {
        if(AdbUtils.disableWifiAdb() == 0) {
          AdbUtils.showNotification(context, false);
          AdbUtils.startWakelockService(context, false);
        }
      }
    } catch (IOException | InterruptedException e) {
      e.printStackTrace();
      Toast.makeText(context, "Could not enable ADB over WiFi.",
    ↪ Toast.LENGTH_LONG).show();
    }
  }
}
```

Listing 16.5: Android boot broadcast receiver (written in Java)

```
1  private boolean devicesConnected(AndroidEmulatorContext emu, PrintStream logger, int
       ↪ adbPort, AbstractBuild<?,?> build)
2  throws IOException, InterruptedException {
3      final ByteArrayOutputStream deviceList = new ByteArrayOutputStream();
4      //Setup a build environment to run the "adb devices" command in.
5      final EnvVars buildEnv = build.getEnvironment(TaskListener.NULL);
6      buildEnv.put("ANDROID_ADB_SERVER_PORT", Integer.toString(adbPort));
7      if (emu.sdk().hasKnownHome()) {
8          buildEnv.put("ANDROID_SDK_HOME", emu.sdk().getSdkHome());
9      }
10     if (emu.launcher().isUnix()) {
11         buildEnv.put("LD_LIBRARY_PATH", String.format("%s/tools/lib",
       ↪ emu.sdk().getSdkRoot()));
12     }
13     //Run the command ADB -d devices
14     Utils.runAndroidTool(emu.launcher(), buildEnv, deviceList, logger, emu.sdk(),
       ↪ Tool.ADB, "-d devices", null);
15     String deviceOutput = deviceList.toString();
16     ArrayList<String> deviceNames = getDeviceNames(deviceOutput, logger);
17     if (deviceNames == null || deviceNames.isEmpty()) {
18         return false;
19     }
20     else {
21         return true;
22     }
23 }
```

Listing 16.6: The devicesConnected method which checks for connected devices (written in Java)

are devices connected and the method returns true. The method `devicesConnected()` is called during the emulator startup method, after the build environment is set up, but before the emulator has been started. With this addition to the plugin, Jenkins will only start an emulator instance if there are no devices attached. This shortens the build time significantly.

16.5 Adapting Job Flow in Jenkins

We need to connect to the physical devices before the emulator plugin runs, so that it will not start any emulator. The flow of the new build process is:

1. Disconnect devices
2. Connect to devices
3. Uninstall apps
4. Run build and tests
5. Disconnect devices
6. Connect to devices
7. Uninstall apps
8. Disconnect devices

We start by disconnecting devices to make sure the connections are up to date. If a device somehow loses connection, it will not appear to have been disconnected unless we specifically disconnect it. After having disconnected any device, we connect to all devices again. When a connection has been established, we uninstall all installed GIRAF apps to make sure that they do not interfere with the test. While we do this after the build itself, we must also do it initially. This is because groups might install GIRAF apps on their devices while disconnected from the test network, and then connect their tablets to the test network later. We also do not want to leave any apps on the devices if groups want to use their tablets after they have been tested on. When we have performed the build we disconnect devices followed by connecting to them — the reason is the same as when we start. Following this we uninstall any GIRAF apps and then disconnect.

Connection Script

To connect to all devices connected to the router we get the information from the HTTP server on the router. The script can be seen in Listing 16.7. We first check the connection to the server in line 3 by supplying the `i` option so that `curl` returns the HTTP response header. We then execute `grep 200` to check that the request was successful (code 200 means a successful connection). After this, we check the error code of that command in line 5. If the HTTP response code was `200`, `grep` sets its exit code to `0`. We then get the ports of all devices connected to the router in line 7. Finally, we simply iterate through those ports and connect to them via ADB (lines 8–11).

Uninstallation Script

The uninstallation script seen in Listing 16.8 uninstalls all GIRAF APKs from all connected devices in parallel. We do it in parallel so that connecting more tablets will not increase

```
#!/bin/bash
URL="http://172.25.11.91/cgi-bin/devices"
curl --connect-timeout 10 -i --silent $URL | head -1 | grep 200 > /dev/null

if [ "$?" -eq "0" ]
then
    DEVICE_PORTS="$(curl --silent $URL)"
    for d in $DEVICE_PORTS
    do
        $ANDROID_HOME/platform-tools/adb connect 172.25.11.91:$d
    done

    sleep 1
else
    echo "No connection found"
fi
```

Listing 16.7: Bash script that connects to devices

the time it takes to uninstall apps. To do it in parallel, we use the GNU parallel tool [61]. In line 14 we get the serial number of all connected devices. In line 19 we run the parallel tool on the function `uninstall`. The variable `$SERIAL_NUMBER` after the three colons is the array that is iterated in parallel. Each element is then passed to the `uninstall` function. The uninstall function gets the packages names of all installed GIRAF apps on the device in line 4. The two `sed` calls at the end make sure they return nothing but the package names, which are to be passed to adb to uninstall the apps. The apps are uninstalled in lines 6–10. The function is exported in line 12 so that parallel can use it.

The script to get the serial numbers of connected devices can be seen in Listing 16.9. It runs `adb devices` and then greps exactly on `device`. `grep -vw emulator` makes sure that no emulators are matched. When an emulator is started in Jenkins, an additional device will appear that is also the emulator. This means that two devices will be shown that are in fact the same. To make sure we do not uninstall twice on the same device, we simply ignore one of the devices listed. When the devices have been found, we remove everything but the serial number itself with `sed 's/\s*device//'` and make sure there is no additional whitespace with `sed 's/\s*//g'`. The `g` at the end makes sure that all matching strings are substituted. For example, the input:

```
List of devices attached
047f671ee24b2839        device
emulator-5554   device
```

will result in the output `047f671ee24b2839`.

Disconnection Script

To disconnect all connected devices in the script shown in Listing 16.10, we simply get the serial number of all connected devices in line 2. We then run a loop that disconnects those serial numbers (lines 4–7).

```
#!/bin/bash
uninstall() {
  # Find all installed giraf apps on this device
  PACKAGE_NAMES="$($ANDROID_HOME/platform-tools/adb -s $1 shell pm list packages -f |
    ↪ grep dk.aau.cs.giraf | sed 's/.*apk=//' | sed 's/\s*//g')"

  for p in $PACKAGE_NAMES
  do
      echo "Uninstalling $p on $1"
      $ANDROID_HOME/platform-tools/adb -s $1 uninstall $p
  done
}
export -f uninstall

SERIAL_NUMBER="$(/srv/scripts/get_serial_numbers.sh)"

# Only uninstall if there is a device
if [ "$SERIAL_NUMBER" ]
then
    parallel uninstall ::: $SERIAL_NUMBER
else
    echo "No device found. Not uninstalling."
fi
```

Listing 16.8: Bash script that uninstalls all installed GIRAF apps on all devices

```
#!/bin/bash
$ANDROID_HOME/platform-tools/adb devices | grep -w device | grep -vw emulator | sed
    ↪ 's/\s*device//' | sed 's/\s*//g'
```

Listing 16.9: Bash script that gets the serial numbers of all connected devices

```
#!/bin/bash
SERIAL_NUMBER="$(/srv/scripts/get_serial_numbers.sh)"

for s in $SERIAL_NUMBER
do
    $ANDROID_HOME/platform-tools/adb disconnect $s
done
```

Listing 16.10: Bash script that disconnects all connected devices

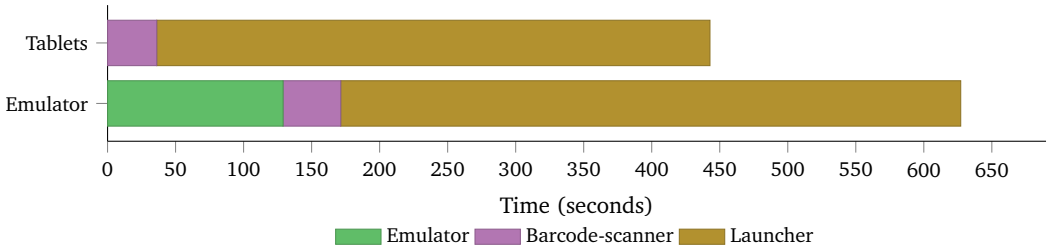


Figure 16.3: Comparison of Launcher build timings using an emulator and tablets

16.6 Evaluation of Build Times With Tablets

We measure the time it takes to build the Launcher job with an emulator and with tablets. The times can be seen in Figure 16.3. As can be seen, building with tablets immediately cuts off roughly two minutes of the build time, as there is no need to start an emulator. In addition building the different components take slightly less time, since testing is not performed on an emulator. The server is also faster because it does not run an emulator concurrently when building jobs. This can be hard to see in the figure. The Launcher component, for example, takes 455 seconds with an emulator, but 406 seconds with a tablet. With the tablet the Launcher job is 29.3 % faster than when building with an emulator.

The exact timings of each task can be seen Appendix A.

17 Improving Monkey Testing

We make some changes to the way monkey tests are performed this sprint. We have a user story to make monkey tests use debug apps and take screenshots of crashes. In addition, due to the changes we have made to make build tests run on physical devices, we have to update monkey tests so that they support those changes.

Chapter Organization This chapter is organized in the following fashion:

- In Section 17.1 we make monkey tests run on debug apps rather than release apps;
- in Section 17.2 we introduce a new Android application which blocks the monkey from accidentally opening the notification bar and thus getting access to settings such as the WiFi connection;
- in Section 17.3 we update monkey tests to be able to run in parallel on all connected devices, because of the addition of physical devices being used by Jenkins now;
- in Section 17.4 we give a summary of the new monkey test flow.

17.1 Running Monkey Tests on Debug Apps

At the moment, monkey tests are run on release versions of the apps. This version uses the production database, which, until now, works fine, because local changes are not persisted in the remote database. However, this is to change as the DB groups work on automatic synchronization between devices and the production database. Obviously, we do not want the monkey tests to insert dummy data into the production database. They could accidentally delete or modify customer data if we allow it to test on the production database. We therefore need to make the monkey test the debug version of the apps instead, as the debug versions automatically use a development database. This corresponds to user story 3.

Prior to running a monkey test, the application to test has to be installed on the device. To make this installation easy, we put the newest version of each compiled debug application on the FTP server using the Bash script shown in Listing 17.1. The overall purpose of the script is to first delete the old version of the application from the FTP directory and then move the new one into the FTP directory. The application file is named using the scheme `[package name]_v[version name]b[build number]_debug_aligned.apk`. To know which file corresponds to a specific application, we need the package name of the application. Part of this is found in line 4 by performing the following steps:

1. The package name is contained in the start of the name of the APK file. We use the `find` command to search for file names in the project directory which end in `_debug_aligned.apk`.

```
1  #!/bin/bash
2
3  FTP_DIR="/srv/ftp/debug_apks/"
4  PACKAGE_W_PATH=$(find . -type f -name "*_debug_aligned.apk" -print | grep
       ↪ ".+(?=_v.+b[0-9]+_debug_aligned\.apk)" -Po)
5  echo "FTP dir: "$FTP_DIR
6
7  if [ -z "$PACKAGE_W_PATH" ]
8  then
9      echo "No file found"
10     exit 1
11 else
12     PACKAGE=$(basename $PACKAGE_W_PATH);
13     echo "Package: $PACKAGE"
14     echo "Remove old files: $FTP_DIR$PACKAGE"*.apk
15     rm "$FTP_DIR$PACKAGE"*.apk
16     find . -type f -name "*_debug_aligned.apk" -print -exec mv {} "$FTP_DIR" \;
17     exit 0
18 fi
```

Listing 17.1: Bash script that moves the debug APK to the FTP server

2. Afterwards, we pipe the file names to the `grep` command and use a regular expression to match anything up to and including the package name. The `P` option specifies the Perl regular expression syntax. The `o` options makes it so that only the matching part of the input string is printed. For example, `grep` with the input `applications/dk.aau.cs.giraf.launcher_v2.4b2_debug_aligned.apk` prints `applications/dk.aau.cs.giraf.launcher` (notice that the last part is the package name).

3. The result of the match is stored in the variable `PACKAGE_W_PATH`

In line 7 we check if an APK matching the naming scheme was found, by checking if `PACKAGE_W_PATH` is empty (the `z` option). If the resulting match is empty we exit, otherwise, we remove the path from the variable `PACKAGE_W_PATH` so we have only the package name left. We use this package name to remove the old APK from the FTP folder before we move the new APK to the FTP folder (lines 12–17).

We add the execution of the move script as a post-build task on all app jobs in Jenkins. We also update the monkey jobs to use the debug versions instead of the release versions. This means that when new apps are built, the debug versions are moved to the debug folder in the FTP directory. Every night, the monkey tests use these debug versions for testing.

17.2 Preventing the Monkey from Changing Settings

We find that the monkey test in some cases opens the notification bar in Android, which provides quick access to various system settings as shown in Figure 17.1. For example,

```
1 adb shell am broadcast -a org.thisisafactory.simiasque.SET_OVERLAY --ez enable true
```

Listing 17.2: Command for enabling the notification bar blocker

Figure 17.1: Settings menu in the notification bar with WiFi-settings, system settings etc.

Figure 17.2: Notification bar blocker running. Notice the blue block in the top of the screen

the monkey test has sometimes disabled the WiFi connection, causing the device to disconnect from the test server. This results in a failed test. To avoid this problem, we find an Android app, *simiasque* [46], which blocks access to the notification bar. The application allows us to block the notification bar using a toggle button. However, this is inconvenient for automated monkey testing, as we do not want to toggle the notification bar blocker manually before and after every monkey test. Because of this, we implement a `BroadcastReceiver` in the app, which allows us to send a message to the application from the server which sets the state of the notification bar blocker. This change is proposed as a pull-request[1] and merged into the project. We can now enable the notification bar using ADB, as shown in Listing 17.2.

[1] https://github.com/Orange-OpenSource/simiasque/pull/1

17.3 Running Monkey Tests in Parallel

With the addition of being able to run tests on multiple physical devices, we need to update monkey tests so that they run in parallel on all connected devices. To do this, we first update our script that installs APKs on a device so that it does so in parallel on all connected devices (see Listing F.1 in Appendix F). We do the same for the script that runs the dummy database insertion app (see Listing F.2 in Appendix F). The modifications to these two scripts are similar to that of the the uninstallation script (Listing 16.8). Finally to run the monkey tests in parallel we make a new script, as the Jenkins emulator plugin does not support this feature. The script can be seen in Listing 17.3. We choose not to extend the Jenkins plugin with this feature, as it is faster for us to make a script.

Before starting the script we send an intent to the Simiasque app that blocks the notification bar in line 11. After running the test we unblock the notifaction bar (line 14).

In line 12, in the `run_monkey` function, we run the monkey test on the device specified as the first argument (`$1`). We place the output in a temporary log file. In line 13 we take a screenshot of the device using the ADB tool [38]. Thus, there will be a screenshot of the device just as it finished running the monkey test, or as the monkey test crashed the app.

Since many devices may be connected, it is important to know on which device a test was performed. To record this information, we get relevant device information in line 16. The `getprop` contains many properties of the device. We select the device information and its SDK version `grep -E "product|sdk|serial"`. Using the `-E` flag enables extended regular expressions allowing us to use the vertical bar character as *or*. We also remove any information matching `ro.boot`, as this is not relevant. Finally the device information is concatenated with the temporary log file into a log file.

Having run the monkey test and stored log files for all devices in line 25 we must check if any of the tests failed. If a test succeeded, the final line of the log file will contain `monkey finished`. If it failed, it will state that it appears to have crashed. Thus we simply check the last line of the log file if it contains `monkey finished`, and if not report a failure. We do this in lines 29–33. We check all log files if they report failure and exit with `1` if any of the log files report failure (line 38).

17.4 Monkey Test Flow

The setup of monkey testing has been through several changes since its inception. The current flow of a monkey test is as follows:

1) Connect to Devices/Run Emulator We start by trying to connect to any physical devices. If there are no devices to connect to, we start an emulator instead.

2) Install APKs The APKs needed for the monkey test are installed. This mostly includes the Launcher app and Pictosearch app, but varies from app to app. The dummy

```
#!/bin/bash
BASE_OUTPUT_NAME=${JOB_NAME}_b${BUILD_NUMBER}

run_monkey() {
    PACKAGE=$2
    BASE_OUTPUT_NAME=$3
    SEED=1
    NO_EVENTS=90000
    DELAY=10 # in ms. 90 000 events with a 10 ms delay is a 15 minute test.

    $ANDROID_HOME/platform-tools/adb -s $1 shell am broadcast -a
    ↪ org.thisisafactory.simiasque.SET_OVERLAY --ez enable true
    $ANDROID_HOME/platform-tools/adb -s $1 shell monkey --kill-process-after-error -v
    ↪ -v -s $SEED --throttle $DELAY -p $PACKAGE $NO_EVENTS >
    ↪ ${BASE_OUTPUT_NAME}_$1.log.tmp
    $ANDROID_HOME/platform-tools/adb -s $1 shell screencap -p | sed 's/\r$//' >
    ↪ ${BASE_OUTPUT_NAME}_$1.png
    $ANDROID_HOME/platform-tools/adb -s $1 shell am broadcast -a
    ↪ org.thisisafactory.simiasque.SET_OVERLAY --ez enable false
    # Get device information
    $ANDROID_HOME/platform-tools/adb -s $1 shell getprop | grep -E
    ↪ "product|sdk|serial" | grep -v ro.boot | cat - ${BASE_OUTPUT_NAME}_$1.log.tmp
    ↪ > ${BASE_OUTPUT_NAME}_$1.log

    rm ${BASE_OUTPUT_NAME}_$1.log.tmp
}
export -f run_monkey

SERIAL_NUMBER="$(/srv/scripts/get_serial_numbers.sh)"
if [ "$SERIAL_NUMBER" ]
then
    parallel run_monkey {1} $1 $BASE_OUTPUT_NAME ::: $SERIAL_NUMBER
    MONKEY_FAILED=false
    for s in $SERIAL_NUMBER
    do
        MONKEY_FINISHED="$(tail -1 ${BASE_OUTPUT_NAME}_${s}.log | grep -i "monkey
    ↪ finished")"
        if ! [ "$MONKEY_FINISHED" ]
        then
            echo "Monkey test on $s appears to have crashed"
            MONKEY_FAILED=true
        fi
    done
    if [ "$MONKEY_FAILED" = true ]
    then
        exit 1
    fi
else
    echo "No device found"
    exit 1
fi
```

Listing 17.3: Bash script that runs monkey tests on all connected devices in parallel

database inserter app and the app to be tested are also installed. The APKs are installed in parallel on all connected devices.

3) Run Dummy Database Inserter The dummy database inserter app is run in parallel on all connected devices after having installed all APKs. We do this so that apps do not have to download the full database which can take 10 minutes or more. The build waits for the dummy database inserter app to finish inserting data before continuing.

4) Enable Simiasque We enable the notification blocker app, Simiasque, to prevent the monkey test from disabling Wi-Fi.

5) Run Monkey Test and Report Any Failures The monkey test is run in parallel on all connected devices. A log file is produced for each device containing relevant device information. A screenshot is also captured immediately after the app has either finished monkey testing or crashed. If a failure occured on any device, a failure is reported.

6) Disable Simiasque Simiasque is disabled so that the notification bar can be used again.

7) Publish Log Files and Screenshots The log files and screenshots are published in Jenkins so that they can be accessed easily.

18 Servicing Developers

This chapter describes our work with the two user stories related to prioritizing jobs in Jenkins, and downloading and installing all apps in an easy way. We have also identified several things the last sprints that can improve usability and avoid some common mistakes among developers with regards to using Git. Even though we do not have a user story to back this up, we decide to implement these minor things to improve the developer-friendliness and to avoid the common mistakes.

Chapter Organization This chapter is organized in the following fashion:

- In Section 18.1 we identify and implement a way of prioritizing jobs in Jenkins, and compare various scheduling algorithms;
- in Section 18.2 we create scripts that developers can execute which will download the newest versions of all apps and install them onto a connected device;
- in Section 18.3 we describe small usability improvements and mechanisms to avoid common mistakes when developers push to Git repositories;
- in Section 18.4 we describe how we document Jenkins targeting future developers of the GIRAF project.

18.1 Jenkins Priority Queue

We have committed ourselves to setup a prioritized build queue in Jenkins, such that libraries have a higher priority than other jobs, such as apps. Developers do not want to wait for libraries to build when there are many apps in the queue. This is because libraries are released on Artifactory often, and waiting for them to be released blocks further development more than waiting for apps to build.

To solve this user story, we install a Jenkins plugin [56] which allows for a prioritized build queue. This plugin allows us to assign each Jenkins job a priority from one to five (where one is the highest priority). We select a scheduling algorithm which manages the way in which jobs are ordered.

18.1.1 Scheduling Algorithms

By introducing a priority queue, a number of potential problems arise. There is a risk of starvation, which means that high priority jobs take all resources such that jobs of lower priority are never run. The overall objective is to increase the average throughput of high priority jobs without making the low priority jobs starve.

The Jenkins plugin comes with three different scheduling algorithms:

Absolute The queue is strictly ordered by priority. The highest priority job is greedily selected every time a new job is to be executed.

Fair Queuing Each priority group has a separate queue. Every time a job is requested the job is added to the queue of the priority group the job belongs to. The scheduler executes builds from each priority group in a round robin manner. This means that each group gets an equal amount of jobs executed, regardless of the distribution of jobs between groups.

Weighted Fair Queuing Each priority has a separate queue and a "bucket". Every time a job is requested, it is added to the queue of the priority the job belongs to and its weight is added to the bucket corresponding to the priority of the job. The scheduler always selects a job from the priority with the lightest bucket. Each priority is weighted, so when a priority five build is enqueued, five times as much weight is added to the corresponding bucket than when a priority one build is enqueued. Because every priority has its own bucket, the jobs with priority five will eventually be started, as the bucket corresponding to priority one will get heavier than the bucket of priority five after five builds with priority one.

The absolute scheduling algorithm clearly improves the throughput of high priority jobs. However, there is a risk of starving jobs. The fair queuing algorithm, on the other hand, does not risk starving jobs, but also does not improve throughput for prioritized jobs. The weighted fair queuing algorithm distributes the resources among the different jobs based on their priorities. The throughput of high priority jobs is increased, and the throughput of low priority tasks is decreased, and the low priority tasks are still guaranteed to be run at some point. Because of this, we use the weighted fair queuing algorithm to make library jobs build faster than other jobs in Jenkins.

We give all libraries priority two by default, and all apps priority four. The Meta-Database library, however, has priority one, as the developers often wait for this specific library to complete.

18.2 Easy App Download and Installation

The user story *Easy Download and Installation of all Apps* describes that developers want to be able to easily install the most recent versions of all apps on their devices. To solve this problem, we develop a script which downloads the most recent apps from the FTP server. The solution must work on both Unix systems and Windows without any additional software.

18.2.1 Keeping Track of Newest Apps

When new applications are built in Jenkins, we upload them to the GIRAF FTP server. We extend this upload functionality to also create a symbolic link to the newest version in a `newest_releases/` directory. This directory contains only one link per app, and thus allows us to avoid finding the newest release in a directory containing all releases. Thus

```
1  #!/bin/bash
2  ftp -i -n ftp://ftpuser:9scWKbP@cs-cust06-int.cs.aau.dk/newest_releases/ << EOF
3      binary
4      mget *
5      quit
6  EOF
7
8  COUNTER=0
9  for f in *.apk
10 do
11     echo "Trying to Uninstall $f"
12     adb uninstall ${f%'.apk'}
13     echo "Installing $f"
14     adb install -r "$f"
15     COUNTER=$(expr $COUNTER + 1)
16 done
17 echo "Installed $COUNTER apps"
18 exit 0
```

Listing 18.1: Bash script that downloads and installs newest APKs

we avoid having to search for the newest APK as we have previously done, simplifying the script. Now, we can simply download all files in this directory to have the most recent versions of all apps.

18.2.2 Downloading and Installing Apps

Because we keep track of the newest apps on the build server, only little work is to be done in the download script. Because of this, we allow ourselves to develop two different scripts: A Bash script for Unix and a Powershell script for Windows. These scripts first connect to the FTP server and downloads the apps. Afterwards, it reinstalls each app on a connected device.

The Unix script can be seen in Listing 18.1. We connect to the FTP with the `-i -n` options that disables prompting when downloading multiple files and disables auto login, respectively (line 2). Being connected we specify that the files to be downloaded are binary, get all files in the newest releases directory and quit (lines 3–5). We then install the downloaded APKs onto a connected device (lines 8–16). The Powershell script can be seen in Appendix E and works similarly.

Because the scripts do not require any additional software and because they download and install all applications automatically, this solution comply with the conditions of satisfaction of the user story.

18.3 Pre-Receive Hook

During the latest sprints, we have discovered several times that developers have committed the database password on Git which causes everyone to change the database

password. This is frustrating and time consuming, and for this reason we choose to implement a pre-receive Git hook which rejects a push containing the database password. This is not a user story but a refactoring of the version control system. A pre-receive hook is run every time something is pushed to the Git repository, but before the changes are applied (in contrast to the post-receive hook previously described). If the script returns a non-zero exit code, the commit is rejected.

As we develop this hook, we add a number of additional features. Overall, it checks for the following:

Database Passwords The database password is stored in a Java file called `Database-Credentials.java`. Every database developer has this file locally and inserts it manually during development. In Jenkins, this file is automatically added to the project during build. Even though the file is part of the gitignore list, it is sometimes commited by mistake. If this file is contained anywhere in the push, it is rejected.

References to Local Dependencies Sometimes, applications reference a `0.0-SNAPSHOT` version of a library, which is used to reference a library which is stored on the developer's computer. When Jenkins builds the job, it breaks because it cannot find the library. Such mistakes are avoided by checking for local dependency references.

Dynamic Dependency References Dependencies can be declared with a dynamic version number (e.g. `1.4.+`). As described earlier, we discourage this as it removes the ability to backtrack exact dependency versions in a specific revision of a repository. We therefore check for such references as well and rejects the push of they are referenced.

Push Enabled The script checks if there exists a file called `git_disabled` in the Jenkins home folder. If the file exists, we instantly reject the push. The contents of the file is returned to the developer when they try to push. One should write a message detailing why pushing is disabled in the file. This allows us to temporarily disable pushes to Git if we for example are restarting Jenkins, and let the developers know why it is disabled.

The full pre-receive Git hook code can be seen in Appendix D.2.

18.4 Documenting Jenkins

Next year the development of the multi-project will be taken over by the current 4th semester software students. To ease the handover of the multi-project we describe the Jenkins configuration, and installation in a reference document. The document can be seen in Appendix I. The document is an attempt to document all of our tacit knowledge of the Jenkins server in a complete record. The document includes how jobs are configured, general management of Jenins, and a description of files used for automation.

19 Sprint Review

We have accomplished all the selected tasks in this sprint. The report tasks which were postponed are related to finishing the parts of the report which are not directly related to the work we did this sprint. As written, these report tasks will be completed in the week between the end of the sprint and the project deadline.

Chapter Organization This chapter is organized in the following fashion:

- In Section 19.1 we evaluate how the sprint went and whether we reached our goals on a group level;
- in Section 19.2 we evaluate how the sprint went on the multi-project level.

19.1 Sprint Goals

Prioritizing Libraries (1) We have changed the build queue in Jenkins to be a prioritized queue, which uses scheduling that favors libraries over apps, but still prevents starvation.

Run monkey tests on debug builds (2) We have changed our monkey test scripts such that the tests now are performed on debug builds of the apps instead of release builds.

Easy download and install (3) We have made a Powershell and a Bash script which download the newest build of every app from the FTP server and install them on the a connected tablet in an automated manner without additional software on one's device.

Document Jenkins configuration (4) We have created a document which details how the jobs are configured as well as which files are important, what they are used for, and where they are. Additionally, the document contains details about the authentication system and recommendations to that. We believe this guide is useful for the future developers.

Screenshot of monkey test (5) We have a user story which required some other changes to how the monkey tests are run. When we made those changes it did not require that much effort to also make the monkey test capture screenshots. Now a screenshot is saved from each device after a monkey test has finished, including if it crashed.

Decrease build times in Jenkins (6) We have improved the build times in Jenkins. Now we have a pool of tablets available for testing, such that we do not have to start

a emulator. This saves time every build. We have decreased the build time by approximately 30 %.

19.2 Multi-Project Sprint Review

The sprint review is held on May 20. There was some stress leading up to the sprint review because of server troubles. We had a server breakdown on May 14 and it was restored on May 18 to the state it was in the morning of May 13. The server was down two work days, which resulted in reduced productivity among all groups. There was no loss of code, because of the decentralized way Git works — the developers still had the code locally. However, the changes made to the configuration of Jenkins during that period was lost. We did not make many changes, so it was quickly restored once the server was up again.

19.2.1 Analysis of Server Breakdown

We analyze the causes of the server breakdown to avoid a similar situation in the future. According to the server responsible group, the problem started when the server has no disk space left. This caused all Git pushes to be rejected and every Jenkins build to fail. The server was granted more space, which was to be added to a specific partition. However, they accidentially corrupted the partition group which caused the server to be unable to mount it. They restored the server to a two-days-old state.

The main lesson to be learned from this experience is that technical problems should be expected. Because we do not have a professional server administrator, we cannot expect configuration changes always to be implemented successfully. We should accommodate this kind of problems by resolving problems in their early stages. We recommend that the server responsible group keep track of the server status and for example set an email notification when the available disk space is low. Ideally the technical operation should be performed at times where developers are not working, for example in weekends. This also allows time to restore the server if something goes wrong.

Because we lost configuration changes in Jenkins, we also recommend to create separate backups of the configuration to somewhere other than the server to prevent such an issue in the future. Other than the Jenkins configuration, nothing was lost during the breakdown.

Conclusion

20 Project Evaluation

The overall goal of this project was to make the GIRAF apps more usable and refined than when we inherited it from last year's students. This goal has been apparent throughout the different parts of the project: From the individual backlog items the groups have worked on, to the refinements made to the development method with build automation and continuous integration. Our personal goal was to improve build automation and the overall testing facilities of the GIRAF project. In this chapter, we will evaluate the project with respect to these goals.

There are several layers to evaluate: The technical work, i.e. the backlog items we completed as well as other tasks we completed by necessity; the development process followed in the group; the accomplishments of the multi-project as a whole, i.e. all individual group's collective progress; and finally the process surrounding the entire multi-project.

Chapter Organization This chapter is organized in the following fashion:

- In Section 20.1 we evaluate the technical work we have done this semester in terms of what user stories we selected and completed;
- in Section 20.2 we evaluate the development method followed internally in our group;
- in Section 20.3 we evaluate the development method employed across the multi-project;
- in Section 20.4 we evaluate the collective progress towards the goal of evolving the GIRAF project into something useful, helping autistic citizens.

20.1 Group Work on User Stories

During this project we have completed 16 backlog items. The result of this work is a significant improvement to the build environment, in particular the continuous integration platform, Jenkins, compared to what we inherited. When we started, the continuous integration platform was just a regular build platform. We changed the configuration such that the platform automatically builds a new version whenever changes are pushed to the master branch of a repository. We also run any unit and UI tests after the build process, and we only consider the build successful if all the tests pass. If the build is successful we also publish the new version. If it is an app, it is published on the alpha track on Google Play. If it is a library the commit message indicates whether to publish a major or minor release, a patch or a development snapshot. Before all these changes, developers manually had to start the builds in Jenkins and no test were run. When we

started there were 13 job configurations on the continuous integration platform, and now we have 42 configurations. We have enabled email notifications so that whenever a build fails, the developer responsible for the bad build gets a notification email with a link to the failed build. We provide statistics like code coverage as well as lint errors and warnings. The automation of the build process gives quick feedback when errors occur. This has helped improved the overall stability of the apps. The build and test automation ensures that frequent integration with the master branch proceeds smoothly, as errors will be spotted immediately.

There are very few tests of the apps in general, so we run monkey tests every night, where the newest version of every app is tested. If an app crashes during a test, a stacktrace is available as well as a screenshot taken the moment the app crashed. The monkey tests were only really functioning quite late in the project and therefore have had limited impact on the overall stability of the apps.

We have worked on decreasing the build time of jobs in Jenkins. We have succeeded in relatively speeding up the build process every time, but we still end up with slower build processes than when we started. This is because a build does more work now than what was done before.

We have also worked on different tasks which make life easier for the developers. In collaboration with the Git responsible group we have removed submodules from Git, which were a hassle for the developers to work with. Instead we have libraries which are pre-compiled and the different versions are specified in the build script. The developers also wanted a easy way of installing the newest version of all apps on a tablet. We have developed scripts which accomplishes this task.

In addition to Jenkins, we have also had the responsibility for the development process in general. During handling these responsibilities, we gather information which is relevant for some or all of the other groups in the multi-project. To make this information available, we use the wiki on Redmine. We have made many guides on different topics. These are:

- UI testing
- Unit testing
- Dependency management
- Multi-project development process
- Continuous integration

As a whole, the developers now work in a simple-to-use environment, which provides many services that help improve the stability of the software.

20.2 Internal Development Method

In our group we have used a physical sprint backlog containing all the tasks to complete in a sprint. These tasks are estimated. In combination with a burndown chart we have been able to track our progress throughout a sprint and prioritize tasks. When we were behind schedule, we were able to identify the least important tasks and remove them

from our sprint backlog. This way, we ensured that we always worked on the most important backlog item. The Daily Scrum meeting has ensured that we consistently updated our sprint backlog and burndown chart. Even though there were times where we got behind schedule, we have been able to adapt and Scrum has worked well for us. The method fits nicely with the Scrum of Scrum method used on the multi-project level, especially the common language of backlog items like user stories has been very useful and makes an effortless transition of backlog items between the subproject and group levels.

20.3 Multi-Project Development Method

The multi-project has been organized according to Scrum of Scrums. We have continually throughout all sprints refined the method to improve our development process. What is described in Chapter 2 is the result of refining the Scrum method to suit our needs. The work we have put into formalizing the development method and the following refinements of it has freed the other groups to dedicate their time to completing their user stories related to more practical work on the GIRAF project. The development method we have defined has brought a close collaboration with the customer which has enabled all groups to reach the GIRAF project goals.

In sprint 1 and to some degree sprint 2, there was some confusion among the groups regarding the development method. We believe this was the reason why some of the meetings ran over time, e.g. the DB sprint 2 planning meeting described in Section 7.1. We therefore had to spend time clarifying this. We underestimated the need for a clear process specification, which we will do if we are to manage a similar project in the future.

At each sprint planning meeting, the requirements of the external customers have been mostly diffuse. Their desires changes over time. For this an agile and adaptive development method is suitable. Part of working agile is collaborating closely, which groups have been good at doing.

We initially kept the product backlog on Redmine. However, this tool proved to be inadequate as the product owners switched to using Google Docs to manage the backlog. The backlog became fragmented, and each product owner had their own backlogs. This made it very hard to find out which backlog items that were being worked on in a given sprint, and made it hard to track progress. The product owners also had to spend time and resources during the last sprint on unifying the product backlog and making sure it was up to date. If there had been a more rigid, perhaps tool assisted, method for organizing and maintaining the product backlog, it would have been easier to get an overview of the heading of the multi-project.

The development process has not been perfect. There have been problems during the project and we have adjusted the method continually to adapt to changing conditions and respond to concerns raised by the developers. However, the method will never be perfect as conditions are bound to change in the future, and the method will have to be adjusted accordingly.

20.4 The Development of GIRAF

When we started working on the GIRAF project, we overtook multiple apps out of which only few were usable. Additionally, the user interface was very inconsistent between the apps, making the GIRAF apps feel and look like independent apps rather than a single suite of apps. It was the main goal of this project to solve these problems by making existing apps stable and consistent.

Throughout the project, every GUI group has solely worked on user stories related to existing apps. The main focus of the initial sprints was bug fixing, and as bugs became resolved, new features started to be developed. Design guidelines were formulated and implemented in the different apps.

The DB groups have written several automated tests in the database libraries to ensure that new changes do not break existing functionality. They have implemented basic synchronization between tablets and generally accommodated structural changes needed by the GUI groups.

The B&D groups have configured a build and test environment which makes it easier for everyone to write automated tests and discover breaking changes in code. They have made every build reproducible, making it easier to reproduce bugs from crash reports. This has had an influence on the stability of the apps.

Overall, every group has been very committed to making the GIRAF project stable and consistent. During sprint reviews, the external customers have indicated a satisfaction with the features we develop. They are satisfied that we develop what they ask for and that the apps look consistent. Therefore, we consider that the multi-project has successfully accomplished the overall goal of the project.

21 Recommendations for Future Developers

As presented in the previous chapter, we have acquired ourselves some experiences from working in a large project setting. To enable the next year's students to learn from these, we make a number of recommendations for future developers that take over the GIRAF project.

Chapter Organization This chapter is organized in the following fashion:

- In Section 21.1 we make recommendations regarding the development method;
- in Section 21.2 we recommend improvements for continuous integration.

21.1 Development Method Recommendations

We recommend that the developers of next year continue using Scrum as their development method. However, we suggest that the subproject structure should be changed such that the subprojects are divided by functionality and not responsibility (see Chapter 12). We also highly recommend using a common product backlog. We suggest finding a tool that eases the management of the product backlog, such that the backlog continues to be unified and is easy to navigate.

We also recommend that there are arranged some social gatherings, outside of the project. In our experience the gatherings are more successful if they are lightweight, otherwise they need to be planned in greater detail and it is hard to find someone who wants to be responsible for the planning. Examples of gatherings could be eating lunch together every Friday, or agreeing to attend the Friday bar. It is easier to work together if you know the people you are working with.

We suggest that roles are distributed among groups. It is very important that groups which fulfill roles that have a large amount of interaction with other groups take that responsibility seriously and are available in person during the day. Otherwise that can be a source of frustration for the other groups. Examples of roles with many interactions are *Git*, *Product Owner* and *Server*.

21.2 Recommendations for Continuous Integration

We have made many improvements to continuous integration and the continuous integration platform Jenkins. There are, however, still improvements to be made:

Automatic Deployment of Scripts Currently, all scripts in the `scriptz` repository used on the server are not automatically moved to their corresponding directories on the server. Whenever a change is made to them in the repository, they have to be manually updated on the server. This is time consuming and error prone. They should instead be automatically deployed.

Automatic Monkey Test Failure Notifications When a monkey test fails, the responsible developers are not automatically notified. Instead they have to manually check the builds or manually subscribe to an RSS feed to get notifications. Monkey tests should be updated so that when a failure occurs, the responsible developers are automatically notified, as is the case for other jobs.

Prevent Push of Snapshots We have implemented a pre-receive Git hook that prevents developers from pushing dynamic dependency versions and local snapshots of libraries, because these can be error prone. We have not prevented the pushing of snapshot versions. Snapshots are the newest version of a library in-between releases and can contain errors. Since we only store the most recent snapshot, using a snapshot version can result in a library version being updated erroneously. Therefore references to snapshot versions should not be pushed to the remote.

Testing There are virtually no tests for the various GIRAF apps and libraries. While we have made improvements to test automation, the code for the apps and libraries is written in such a way that it is very difficult to test. The code should be updated to make testing easy for developers.

Graceful Tablet Disconnection If a tablet is disconnected after a build job is started, the build job will fail even if there are other connected tablets. This should be handled more gracefully by still running tests on the connected tablets. If all devices have been disconnected, an emulator should be started instead.

These items can be found in the product backlog.

Bibliography

[1] Saswat Anand and Mary Jean Harrold. “Heap cloning: Enabling dynamic symbolic execution of java programs”. In: *ASE*. 2011, pp. 33–42.

[2] Saswat Anand et al. “Automated concolic testing of smartphone apps”. In: *SIGSOFT FSE*. 2012, p. 59.

[3] Android. *AAR Format*. [Accessed April 9, 2015]. 2015. URL: http://tools.android.com/tech-docs/new-build-system/aar-format.

[4] Android. *New Build System*. [Accessed April 9, 2015]. 2015. URL: http://tools.android.com/tech-docs/new-build-system.

[5] Apache. *Apache Accumolo Versioning*. [Accessed April 22, 2015]. 2015. URL: https://accumulo.apache.org/versioning.html.

[6] Apache. *APR's Version Numbering*. [Accessed April 22, 2015]. 2015. URL: https://apr.apache.org/versioning.html.

[7] Apache. *Version Policy*. [Accessed April 22, 2015]. 2015. URL: http://isis.apache.org/contributors/versioning-policy.html.

[8] The Apache Software Foundation. *Apache Maven*. [Accessed March 3, 2015]. 2015. URL: https://maven.apache.org/.

[9] Apache Archiva. *Apache Archiva: The Build Artifact Repository Manager*. [Accessed April 16, 2015]. 2015. URL: http://archiva.apache.org/index.cgi.

[10] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004. ISBN: 0-321-27865-8.

[11] Kent Beck et al. *Manifesto for Agile Software Development*. [Accessed April 15, 2015]. 2001. URL: http://www.agilemanifesto.org/.

[12] Christian Becker and Björn Hurling. *gradle-play-publisher*. [Accessed March 31, 2015]. 2015. URL: https://github.com/Triple-T/gradle-play-publisher.

[13] Colin Bird and Rachel Davies. *Scaling Scrum, Scrum Gathering London 2007, slides*. 2007. URL: https://www.scrumalliance.org/resource_download/287.

[14] Brian. *How can I connect to Android with ADB over TCP?* [Accessed May 14, 2015]. 2010. URL: http://stackoverflow.com/a/3623727/236130.

[15] Mike Cohn. *Succeeding with Agile*. Addison-Wesley Professional, 2009. ISBN: 0-321-66053-6.

[16] Mike Cohn. *User Stories Applied: For Agile Software Development*. Addison-Wesley Professional, 2004. ISBN: 0-321-20568-5.

[17] DD-WRT. *DD-WRT*. [Accessed May 18, 2015]. 2015. URL: http://dd-wrt.com.

[18] Doxygen. *Doxygen*. [Accessed February 27, 2015]. 2015. URL: http://www.stack.nl/~dimitri/doxygen/.

[19] EclEmma. *JaCoCo Java Code Coverage Library*. [Accessed April 9, 2015]. 2015. URL: http://www.eclemma.org/jacoco/.

[20] Martin Fowler. *Continuous Integration*. [Accessed March 4, 2015]. 2006. URL: http://www.martinfowler.com/articles/continuousIntegration.html.

[21] Martin Fowler. *FeatureBranch*. [Accessed March 4, 2015]. 2009. URL: http://martinfowler.com/bliki/FeatureBranch.html.

[22] Martin Fowler. *PendingHead*. [Accessed March 4, 2015]. 2007. URL: http://martinfowler.com/bliki/PendingHead.html.

[23] Martin Fowler. *ReproducibleBuild*. [Accessed April 9, 2015]. 2010. URL: http://martinfowler.com/bliki/ReproducibleBuild.html.

[24] Git. *3.4 Git Branching - Branching Workflows*. [Accessed May 8, 2015]. 2015. URL: http://git-scm.com/book/en/v2/Git-Branching-Branching-Workflows.

[25] Git. *Customizing Git — Git Attributes*. [Accessed March 3, 2015]. 2015. URL: http://git-scm.com/book/en/v2/Customizing-Git-Git-Attributes.

[26] Git. *Git*. [Accessed March 3, 2015]. 2015. URL: http://www.git-scm.com/.

[27] Git. *Git Tools — Submodules*. [Accessed April 9, 2015]. 2015. URL: http://git-scm.com/book/be/v2/Git-Tools-Submodules.

[28] Boris Gloger. *Your Scrum Checklist: Scrum Hard Facts*. InfoQ.com, 2010. ISBN: 978-3-000-32112-2.

[29] Google. *Android Debug Bridge*. [Accessed May 14, 2015]. 2015. URL: http://developer.android.com/tools/help/adb.html.

[30] Google. *Lint | Android Developers*. [Accessed February 20, 2015]. 2015. URL: http://developer.android.com/tools/help/lint.html.

[31] Google. *Testing | Android Developers*. [Accessed February 26, 2015]. 2015. URL: http://developer.android.com/tools/testing/index.html.

[32] Gradle. *Dependency Management*. [Accessed April 16, 2015]. 2015. URL: https://gradle.org/docs/current/userguide/dependency_management.html.

[33] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010. ISBN: 978-0-321-67025-0.

[34] Michael Hüttermann. *Mastering Binaries with Hudson, Maven, Git, Artifactory, and Bintray*. [Accessed April 15, 2015]. 2014. URL: http://www.oracle.com/technetwork/articles/java/enterprise-binaries-2227134.html.

[35] Jenkins community. *Jenkins CI*. [Accessed February 20, 2015]. 2015. URL: http://jenkins-ci.org.

[36] JFrog. *Artifactory — The Open Source Maven Repository Manager*. [Accessed April 16, 2015]. 2015. URL: http://www.jfrog.com/open-source.

[37] Kohsuke Kawaguchi et al. *JaCoCo Plugin*. [Accessed April 9, 2015]. 2015. URL: https://wiki.jenkins-ci.org/display/JENKINS/JaCoCo+Plugin.

[38] kev. *How to capture the screen as fast as possible through adb?* [Accessed May 17, 2015]. 2014. URL: http://stackoverflow.com/questions/13984017/how-to-capture-the-screen-as-fast-as-possible-through-adb#comment33165978_18518942.

[39] Craig Larman. *Agile Iterative Development: A Manager's Guide*. Addison-Wesley Professional, 2003. Chap. 7. ISBN: 0-13-111155-8.

[40] Mikaël Mayer. *Shell script: find maximum value in a sequence of integers without sorting*. [Accessed May 15, 2015]. 2012. URL: http://stackoverflow.com/a/11931715.

[41] Mountain Goat Software. *Scrum Team*. [Accessed April 16, 2015]. 2015. URL: https://www.mountaingoatsoftware.com/agile/scrum/team.

[42] Mountain Goat Software. *Product Backlog*. [Accessed April 24, 2015]. 2015. URL: https://www.mountaingoatsoftware.com/agile/scrum/product-backlog.

[43] Scott M. Myers, Chris Plauché Johnson, and the Council on Children With Disabilities. "Management of Children With Autism Spectrum Disorders". In: *Pediatrics* 120.5 (2007), pp. 1162–1182. URL: http://pediatrics.aappublications.org/content/120/5/1162.abstract.

[44] The National Autistic Society. *Diagnosis: the process for adults*. [Accessed May 20, 2015]. 2015. URL: http://www.autism.org.uk/about-autism/all-about-diagnosis/diagnosis-information-for-adults/how-do-i-get-a-diagnosis.aspx.

[45] The National Autistic Society. *Diagnosis: the process for children*. [Accessed May 20, 2015]. 2015. URL: http://www.autism.org.uk/About-autism/All-about-diagnosis/Diagnosis-the-process-for-children.aspx.

[46] Orange OpenSource. *Simiasque*. [Accessed May 19, 2015]. 2015. URL: https://github.com/Orange-OpenSource/simiasque.

[47] OpenWRT. *OpenWRT*. [Accessed May 18, 2015]. 2015. URL: http://openwrt.org.

[48] Oracle. *Javadoc*. [Accessed February 27, 2015]. 2015. URL: http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html.

[49] Christopher Orr. *Android Emulator Plugin*. [Accessed May 15, 2015]. 2015. URL: https://wiki.jenkins-ci.org/display/JENKINS/Android+Emulator+Plugin.

[50] Christopher Orr. *Google Play Android Publisher Plugin*. [Accessed March 31, 2015]. 2015. URL: https://wiki.jenkins-ci.org/display/JENKINS/Google+Play+Android+Publisher+Plugin.

[51] Tom Preston-Werner. *Semantic Versioning 2.0.0*. [Accessed April 22, 2015]. 2015. URL: http://semver.org/.

[52] Chenxiong Qian. *Have a look at Acteve — Run Acteve*. [Accessed March 13, 2015]. 2013. URL: http://chenxiong.blogspot.nl/2013/10/have-look-at-acteve-run-acteve.html.

[53] Redmine. *Redmine.org*. [Accessed April 15, 2015]. 2015. URL: http://www.redmine.org/.

[54] Robolectric. *Robolectric*. [Accessed April 24, 2015]. 2015. URL: http://www.robolectric.org/.

[55] K.S. Rubin. *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. Addison-Wesley Signature Series (Cohn). Pearson Education, 2012. ISBN: 978-0-321-70037-7. URL: https://books.google.com/books?id=3vGEcOfCkdwC.

[56] Magnus Sandberg. *Priority Sorter Plugin*. [Accessed April 6, 2015]. 2015. URL: https://wiki.jenkins-ci.org/display/JENKINS/Priority+Sorter+Plugin.

[57] Martin Schroeder. *inheritance-plugin*. [Accessed April 16, 2015]. 2015. URL: https://wiki.jenkins-ci.org/display/JENKINS/inheritance-plugin.

[58] Ieee Computer Society, P. Bourque, and R.E. Fairley. *Guide to the Software Engineering Body of Knowledge (Swebok®): Version 3.0*. IEEE Computer Society Press, 2014. ISBN: 978-0-7695-5166-1.

[59] Sonatype. *Sonatype Nexus*. [Accessed April 16, 2015]. 2015. URL: http://www.sonatype.org/nexus/.

[60] Andrew S. Tanenbaum. *Modern Operating Systems*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN: 9780136006633.

[61] O. Tange. "GNU Parallel - The Command-Line Power Tool". In: *;login: The USENIX Magazine* 36.1 (Feb. 2011), pp. 42–47. DOI: 10.5281/zenodo.16303. URL: http://www.gnu.org/s/parallel.

[62] Wikipedia. *Concolic testing*. [Accessed February 26, 2015]. 2015. URL: http://en.wikipedia.org/wiki/Concolic_testing.

A Exact Build Times of Launcher App

A.1 Before New Dependency System

The run targets are: `check`, `connectedCheck`, and `assembleRelease`. The time of all tasks can be seen in Listing A.1. Notice that the emulator time for both the fast and slow are included, with the slow in parentheses.

```
18000ms;Emulator_fast
(115000; Emulator_slow)
309ms;GIRAF_Components:preBuild
105ms;OasisLib:mergeReleaseProguardFiles
94ms;OasisLib:preBuild
154ms;local-db:preBuild
63ms;local-db:compileReleaseAidl
86ms;metadata_local:compileJava
60ms;metadata_local:jar
65ms;local-db:compileReleaseJava
1015ms;local-db:packageReleaseJar
63ms;metadata:compileJava
92ms;OasisLib:compileReleaseJava
50ms;GIRAF_Components:prepareWorkspaceOasisLibUnspecifiedLibrary
238ms;GIRAF_Components:mergeDebugResources
200ms;GIRAF_Components:processDebugResources
114ms;GIRAF_Components:compileDebugJava
243ms;GIRAF_Components:mergeReleaseResources
171ms;GIRAF_Components:processReleaseResources
53ms;GIRAF_Components:compileReleaseJava
11826ms;GIRAF_Components:lint
60ms;OasisLib:compileDebugJava
3872ms;OasisLib:lint
128ms;barcodescanner:preBuild
102ms;barcodescanner:processDebugResources
161ms;barcodescanner:compileDebugJava
72ms;barcodescanner:packageReleaseResources
67ms;barcodescanner:processReleaseResources
69ms;barcodescanner:compileReleaseJava
6026ms;barcodescanner:lint
207ms;GIRAF_Components:packageReleaseResources
116ms;GIRAF_Components:bundleRelease
110ms;launcher-app:preBuild
89ms;launcher-app:prepareWorkspaceGIRAF_ComponentsUnspecifiedLibrary
348ms;launcher-app:mergeDebugResources
175ms;launcher-app:processDebugResources
59ms;launcher-app:compileDebugJava
130ms;launcher-app:mergeReleaseResources
```

```
87ms;launcher-app:processReleaseResources
50ms;launcher-app:compileReleaseJava
7292ms;launcher-app:lint
976ms;local-db:lint
81ms;GIRAF_Components:packageDebugResources
81ms;GIRAF_Components:mergeDebugTestResources
91007ms;GIRAF_Components:connectedAndroidTest
139ms;OasisLib:unzipJacocoAgent
73ms;OasisLib:instrumentDebug
56ms;OasisLib:packageDebugResources
98ms;OasisLib:compileDebugTestJava
59ms;OasisLib:packageDebugTest
152473ms;OasisLib:connectedAndroidTest
2376ms;OasisLib:createDebugCoverageReport
53ms;launcher-app:dexDebug
14950ms;local-db:connectedAndroidTest
173ms;GIRAF_Components:preBuild
111ms;OasisLib:mergeReleaseProguardFiles
81ms;OasisLib:preBuild
92ms;local-db:preBuild
71ms;metadata_local:compileJava
186ms;GIRAF_Components:mergeReleaseResources
204ms;GIRAF_Components:processReleaseResources
64ms;GIRAF_Components:compileReleaseJava
184ms;GIRAF_Components:packageReleaseResources
117ms;GIRAF_Components:bundleRelease
139ms;barcodescanner:preBuild
66ms;barcodescanner:processReleaseResources
881ms;barcodescanner:compileReleaseJava
81ms;launcher-app:preBuild
69ms;launcher-app:prepareWorkspaceGIRAF_ComponentsUnspecifiedLibrary
191ms;launcher-app:mergeReleaseResources
96ms;launcher-app:processReleaseResources
6652ms;launcher-app:lintVitalRelease
89ms;launcher-app:dexRelease
```

Listing A.1: Running time of tasks of old dependency system

A.2 After New Dependency System

The build time of the Launcher app with the targets `check`, `connectedCheck`, and `assembleRelease` can be seen in in Listing A.2.

The build time with the targets `clean`, `check`, `connectedCheck`, `increaseVersionCode`, `assembleRelease`, and `publishApkRelease` can be seen in Listing A.3.

```
115000ms;emulator
357ms;barcodescanner:preBuild
210ms;barcodescanner:compileDebugAidl
157ms;barcodescanner:compileDebugRenderscript
107ms;barcodescanner:packageDebugResources
77ms;barcodescanner:processDebugManifest
108ms;barcodescanner:processDebugResources
```

```
161ms;barcodescanner:compileDebugJava
61ms;barcodescanner:compileReleaseRenderscript
109ms;barcodescanner:packageReleaseResources
107ms;barcodescanner:processReleaseResources
90ms;barcodescanner:compileReleaseJava
15569ms;barcodescanner:lint
104ms;barcodescanner:packageReleaseJar
115ms;launcher-app:preBuild
213ms;launcher-app:prepareDkAauCsGirafGirafComponent10Library
518ms;launcher-app:mergeDebugResources
159ms;launcher-app:processDebugResources
88ms;launcher-app:compileDebugJava
59ms;launcher-app:renameApk
109ms;launcher-app:generateReleaseBuildConfig
367ms;launcher-app:mergeReleaseResources
527ms;launcher-app:processReleaseManifest
852ms;launcher-app:processReleaseResources
3113ms;launcher-app:compileReleaseJava
11463ms;launcher-app:lint
71ms;barcodescanner:processDebugTestResources
99ms;barcodescanner:connectedAndroidTest
67ms;launcher-app:dexDebug
2181ms;launcher-app:packageDebug
227ms;launcher-app:zipalignDebug
62696ms;launcher-app:connectedAndroidTest
55ms;barcodescanner:mergeReleaseProguardFiles
209ms;barcodescanner:preBuild
131ms;barcodescanner:compileReleaseRenderscript
99ms;barcodescanner:packageReleaseResources
57ms;barcodescanner:processReleaseManifest
90ms;barcodescanner:processReleaseResources
113ms;barcodescanner:compileReleaseJava
133ms;barcodescanner:packageReleaseJar
68ms;launcher-app:preBuild
141ms;launcher-app:prepareDkAauCsGirafGirafComponent10Library
60ms;launcher-app:renameApk
101ms;launcher-app:mergeReleaseAssets
491ms;launcher-app:mergeReleaseResources
208ms;launcher-app:processReleaseResources
73ms;launcher-app:compileReleaseJava
9256ms;launcher-app:lintVitalRelease
4864ms;launcher-app:dexRelease
2035ms;launcher-app:packageRelease
226ms;launcher-app:zipalignRelease
```

Listing A.2: Running time of tasks of new dependency system with limited targets

```
115000ms;emulator
174ms;barcodescanner:clean
433ms;launcher-app:clean
205ms;barcodescanner:preBuild
245ms;barcodescanner:compileDebugAidl
151ms;barcodescanner:compileDebugRenderscript
54ms;barcodescanner:generateDebugBuildConfig
```

```
77ms;barcodescanner:mergeDebugAssets
1950ms;barcodescanner:packageDebugResources
434ms;barcodescanner:processDebugManifest
376ms;barcodescanner:processDebugResources
4731ms;barcodescanner:compileDebugJava
88ms;barcodescanner:compileReleaseRenderscript
55ms;barcodescanner:mergeReleaseAssets
870ms;barcodescanner:packageReleaseResources
131ms;barcodescanner:processReleaseManifest
272ms;barcodescanner:processReleaseResources
2166ms;barcodescanner:compileReleaseJava
10758ms;barcodescanner:lint
439ms;barcodescanner:packageReleaseJar
171ms;barcodescanner:bundleRelease
64ms;launcher-app:preBuild
174ms;launcher-app:prepareComAndroidSupportSupportV42103Library
993ms;launcher-app:prepareDkAauCsGirafGirafComponent10Library
56ms;launcher-app:prepareDkAauCsGirafLocalDb10Library
96ms;launcher-app:prepareFrAvianeyComViewpagerindicatorLibrary241Library
133ms;launcher-app:prepareWorkspaceBarcodescannerUnspecifiedLibrary
19751ms;launcher-app:mergeDebugResources
161ms;launcher-app:processDebugManifest
744ms;launcher-app:processDebugResources
1298ms;launcher-app:compileDebugJava
19216ms;launcher-app:mergeReleaseResources
118ms;launcher-app:processReleaseManifest
565ms;launcher-app:processReleaseResources
638ms;launcher-app:compileReleaseJava
9303ms;launcher-app:lint
174ms;barcodescanner:packageDebugJar
226ms;barcodescanner:bundleDebug
74ms;barcodescanner:processDebugTestManifest
51ms;barcodescanner:mergeDebugTestAssets
401ms;barcodescanner:mergeDebugTestResources
485ms;barcodescanner:processDebugTestResources
130ms;barcodescanner:compileDebugTestJava
7823ms;barcodescanner:preDexDebugTest
1882ms;barcodescanner:dexDebugTest
804ms;barcodescanner:packageDebugTest
79ms;barcodescanner:connectedAndroidTest
50528ms;launcher-app:preDexDebug
5482ms;launcher-app:dexDebug
1567ms;launcher-app:packageDebug
237ms;launcher-app:zipalignDebug
87ms;launcher-app:processDebugTestResources
94ms;launcher-app:compileDebugTestJava
713ms;launcher-app:dexDebugTest
51ms;launcher-app:packageDebugTest
60651ms;launcher-app:connectedAndroidTest
97ms;launcher-app:increaseVersionCode
235ms;barcodescanner:preBuild
88ms;barcodescanner:compileReleaseRenderscript
52ms;barcodescanner:packageReleaseResources
60ms;barcodescanner:processReleaseResources
```

```
97ms;barcodescanner:compileReleaseJava
68ms;barcodescanner:packageReleaseJar
59ms;barcodescanner:bundleRelease
62ms;launcher-app:preBuild
124ms;launcher-app:prepareDkAauCsGirafGirafComponent10Library
81ms;launcher-app:renameApk
58ms;launcher-app:generateReleaseBuildConfig
294ms;launcher-app:mergeReleaseResources
455ms;launcher-app:processReleaseManifest
1077ms;launcher-app:processReleaseResources
3421ms;launcher-app:compileReleaseJava
8860ms;launcher-app:lintVitalRelease
57ms;launcher-app:compileReleaseNdk
329ms;launcher-app:preDexRelease
4737ms;launcher-app:dexRelease
2041ms;launcher-app:packageRelease
146ms;launcher-app:zipalignRelease
11322ms;launcher-app:publishApkRelease
```

Listing A.3: Running time of tasks of new dependency system with full targets

A.3 Before Tablets

The build time of the Launcher app before using tablets can be seen in Listing A.4.

```
129000ms;emulator
263ms;barcodescanner:clean
676ms;launcher-app:clean
193ms;barcodescanner:preBuild
193ms;barcodescanner:compileDebugAidl
157ms;barcodescanner:compileDebugRenderscript
206ms;barcodescanner:mergeDebugAssets
1954ms;barcodescanner:packageDebugResources
348ms;barcodescanner:processDebugManifest
336ms;barcodescanner:processDebugResources
4795ms;barcodescanner:compileDebugJava
52ms;barcodescanner:compileReleaseAidl
92ms;barcodescanner:compileReleaseRenderscript
50ms;barcodescanner:mergeReleaseAssets
1154ms;barcodescanner:packageReleaseResources
181ms;barcodescanner:processReleaseManifest
404ms;barcodescanner:processReleaseResources
1957ms;barcodescanner:compileReleaseJava
13194ms;barcodescanner:lint
447ms;barcodescanner:packageReleaseJar
60ms;barcodescanner:compileReleaseNdk
250ms;barcodescanner:bundleRelease
113ms;launcher-app:preBuild
155ms;launcher-app:prepareComAndroidSupportSupportV42103Library
1008ms;launcher-app:prepareDkAauCsGirafGirafComponent1040Library
465ms;launcher-app:prepareDkAauCsGirafLocalDb514Library
95ms;launcher-app:prepareDkAauCsGirafOasisLib900Library
135ms;launcher-app:prepareDkAauCsGirafShowcaseView100Library
```

```
91ms;launcher-app:prepareFrAvianeyComViewpagerindicatorLibrary241Library
169ms;launcher-app:prepareWorkspaceBarcodescannerUnspecifiedLibrary
56ms;launcher-app:mergeDebugAssets
29760ms;launcher-app:mergeDebugResources
285ms;launcher-app:processDebugManifest
1223ms;launcher-app:processDebugResources
1722ms;launcher-app:compileDebugJava
26897ms;launcher-app:mergeReleaseResources
132ms;launcher-app:processReleaseManifest
784ms;launcher-app:processReleaseResources
1024ms;launcher-app:compileReleaseJava
27874ms;launcher-app:mergeUnsignedBuildResources
288ms;launcher-app:processUnsignedBuildManifest
971ms;launcher-app:processUnsignedBuildResources
869ms;launcher-app:compileUnsignedBuildJava
15331ms;launcher-app:lint
179ms;barcodescanner:packageDebugJar
162ms;barcodescanner:bundleDebug
111ms;barcodescanner:processDebugTestManifest
442ms;barcodescanner:mergeDebugTestResources
431ms;barcodescanner:processDebugTestResources
63ms;barcodescanner:compileDebugTestJava
9341ms;barcodescanner:preDexDebugTest
1914ms;barcodescanner:dexDebugTest
1175ms;barcodescanner:packageDebugTest
152ms;barcodescanner:connectedAndroidTest
92ms;launcher-app:unzipJacocoAgent
1651ms;launcher-app:instrumentDebug
105559ms;launcher-app:dexDebug
3783ms;launcher-app:packageDebug
625ms;launcher-app:zipalignDebug
284ms;launcher-app:prepareComAndroidSupportTestEspressoEspressoCore20Library
59ms;launcher-app:processDebugTestManifest
395ms;launcher-app:processDebugTestResources
476ms;launcher-app:compileDebugTestJava
24672ms;launcher-app:dexDebugTest
934ms;launcher-app:packageDebugTest
77797ms;launcher-app:connectedAndroidTest
1918ms;launcher-app:createDebugCoverageReport
113ms;launcher-app:increaseVersionCode
74ms;barcodescanner:mergeReleaseProguardFiles
561ms;barcodescanner:preBuild
62ms;barcodescanner:compileReleaseAidl
129ms;barcodescanner:compileReleaseRenderscript
945ms;barcodescanner:packageReleaseResources
60ms;barcodescanner:processReleaseResources
174ms;barcodescanner:compileReleaseJava
126ms;barcodescanner:packageReleaseJar
66ms;launcher-app:preBuild
199ms;launcher-app:prepareDkAauCsGirafGirafComponent1040Library
96ms;launcher-app:prepareWorkspaceBarcodescannerUnspecifiedLibrary
77ms;launcher-app:generateReleaseBuildConfig
458ms;launcher-app:mergeReleaseResources
537ms;launcher-app:processReleaseManifest
```

```
2068ms;launcher-app:processReleaseResources
4022ms;launcher-app:compileReleaseJava
9550ms;launcher-app:lintVitalRelease
58ms;launcher-app:compileReleaseNdk
95341ms;launcher-app:dexRelease
3652ms;launcher-app:packageRelease
296ms;launcher-app:zipalignRelease
10443ms;launcher-app:publishApkRelease
```

Listing A.4: Running time of tasks before using tablets

A.4 After Tablets

The build time of the Launcher app after using tablets can be seen in Listing A.5.

```
0ms;emulator
96ms;barcodescanner:clean
353ms;launcher-app:clean
873ms;barcodescanner:preBuild
139ms;barcodescanner:compileDebugAidl
145ms;barcodescanner:compileDebugRenderscript
65ms;barcodescanner:generateDebugBuildConfig
134ms;barcodescanner:mergeDebugAssets
1491ms;barcodescanner:packageDebugResources
239ms;barcodescanner:processDebugManifest
352ms;barcodescanner:processDebugResources
3579ms;barcodescanner:compileDebugJava
52ms;barcodescanner:compileReleaseAidl
63ms;barcodescanner:compileReleaseRenderscript
814ms;barcodescanner:packageReleaseResources
105ms;barcodescanner:processReleaseManifest
434ms;barcodescanner:processReleaseResources
1290ms;barcodescanner:compileReleaseJava
10690ms;barcodescanner:lint
793ms;barcodescanner:packageReleaseJar
61ms;barcodescanner:compileReleaseNdk
159ms;barcodescanner:bundleRelease
80ms;launcher-app:preBuild
100ms;launcher-app:prepareComAndroidSupportSupportV42103Library
817ms;launcher-app:prepareDkAauCsGirafGirafComponent1040Library
208ms;launcher-app:prepareDkAauCsGirafLocalDb514Library
100ms;launcher-app:prepareWorkspaceBarcodescannerUnspecifiedLibrary
26272ms;launcher-app:mergeDebugResources
181ms;launcher-app:processDebugManifest
795ms;launcher-app:processDebugResources
1351ms;launcher-app:compileDebugJava
25953ms;launcher-app:mergeReleaseResources
193ms;launcher-app:processReleaseManifest
821ms;launcher-app:processReleaseResources
823ms;launcher-app:compileReleaseJava
27884ms;launcher-app:mergeUnsignedBuildResources
155ms;launcher-app:processUnsignedBuildManifest
940ms;launcher-app:processUnsignedBuildResources
```

```
821ms;launcher-app:compileUnsignedBuildJava
16594ms;launcher-app:lint
142ms;barcodescanner:packageDebugJar
174ms;barcodescanner:bundleDebug
139ms;barcodescanner:processDebugTestManifest
52ms;barcodescanner:mergeDebugTestAssets
433ms;barcodescanner:mergeDebugTestResources
486ms;barcodescanner:processDebugTestResources
59ms;barcodescanner:compileDebugTestJava
9278ms;barcodescanner:preDexDebugTest
1273ms;barcodescanner:dexDebugTest
615ms;barcodescanner:packageDebugTest
992ms;launcher-app:instrumentDebug
97612ms;launcher-app:dexDebug
2710ms;launcher-app:packageDebug
455ms;launcher-app:zipalignDebug
195ms;launcher-app:prepareComAndroidSupportTestEspressoEspressoCore20Library
135ms;launcher-app:processDebugTestResources
575ms;launcher-app:compileDebugTestJava
23737ms;launcher-app:dexDebugTest
820ms;launcher-app:packageDebugTest
39042ms;launcher-app:connectedAndroidTest
1377ms;launcher-app:createDebugCoverageReport
93ms;launcher-app:increaseVersionCode
70ms;barcodescanner:mergeReleaseProguardFiles
404ms;barcodescanner:preBuild
63ms;barcodescanner:compileReleaseAidl
164ms;barcodescanner:compileReleaseRenderscript
802ms;barcodescanner:mergeReleaseAssets
77ms;barcodescanner:packageReleaseResources
60ms;barcodescanner:processReleaseResources
111ms;barcodescanner:compileReleaseJava
162ms;barcodescanner:packageReleaseJar
102ms;launcher-app:preBuild
240ms;launcher-app:prepareDkAauCsGirafGirafComponent1040Library
54ms;launcher-app:prepareWorkspaceBarcodescannerUnspecifiedLibrary
55ms;launcher-app:compileReleaseRenderscript
104ms;launcher-app:renameApk
86ms;launcher-app:generateReleaseBuildConfig
413ms;launcher-app:mergeReleaseResources
514ms;launcher-app:processReleaseManifest
1085ms;launcher-app:processReleaseResources
3806ms;launcher-app:compileReleaseJava
11950ms;launcher-app:lintVitalRelease
125ms;launcher-app:compileReleaseNdk
102023ms;launcher-app:dexRelease
3471ms;launcher-app:packageRelease
159ms;launcher-app:zipalignRelease
10287ms;launcher-app:publishApkRelease
```

Listing A.5: Running time of tasks after using tablets

B Gradle Plugins

The plugin to help publish APKs can be seen in Listing B.1. The plugin to publish libraries can be seen in Listing B.2.

```
1  package dk.giraf.gradle
2
3  import org.gradle.api.Plugin
4  import org.gradle.api.Project
5  import org.gradle.api.GradleException
6  import java.util.regex.Pattern
7
8  class DeployPlugin implements Plugin<Project> {
9
10   def manifestFilePath = 'src/main/AndroidManifest.xml'
11   def versionCodesFilePath = '/srv/jenkins/project_version_codes/versionCodes.
       ↪ properties'
12   def keyStorePath = '/srv/jenkins/google_play_keys/keystore.properties'
13
14   void apply(Project project) {
15     //Create play account extension
16     project.extensions.create ('playAcc', GooglePlayAccount)
17
18     project.task('renameApk') << {
19       // Set custom apk name
20       project.android.applicationVariants.all { variant ->
21         def versionCode = getVersionCode(project, applicationId)['value']
22
23         def applicationId = project.android.applicationVariants.applicationId[0]
24         def name
25         variant.outputs.each { output ->
26           def apkDirectory = output.outputFile.parentFile
27           def manifestParser = new com.android.builder.core.DefaultManifestParser()
28
29           if (output.zipAlign) {
30             name = applicationId + "_v" + manifestParser.getVersionName(project.
       ↪ android.sourceSets.main.manifest.srcFile) + "b" + versionCode + "_" + variant.
       ↪ buildType.name.toLowerCase() + "_aligned.apk"
31             output.outputFile = new File(apkDirectory, name)
32           }
33
34           name = applicationId + "_v" + manifestParser.getVersionName(project.android.
       ↪ sourceSets.main.manifest.srcFile) + "b" + versionCode + "_" + variant.
       ↪ buildType.name.toLowerCase() + "_unaligned.apk"
35           output.packageApplication.outputFile = new File(apkDirectory, name)
36         }
37       }
38     }
39     // Autoincrement version code in android manifest
40     project.task('increaseVersionCode') << {
```

```
      def applicationId = project.android.applicationVariants.applicationId[0]
      // Open version codes property file
      if (project.file(versionCodesFilePath).exists() != true) {
        throw new GradleException("No version code file found. Only jenkins should run
    ↪  this task")
      }

      // Open manifest file and find versionCode
      def manifestFile = project.file(manifestFilePath)
      def pattern = Pattern.compile("versionCode=\"(\\d+)\"")
      def manifestText = manifestFile.getText()
      def matcher = pattern.matcher(manifestText)
      matcher.find()

      def newVersion = getVersionCode(project, applicationId)
      def versionCode = newVersion['value']
      if (!newVersion['created']) {
        // Increment version code if not new
        versionCode++
        // Write version code to manifest
        def manifestContent = matcher.replaceAll("versionCode=\"" + versionCode + "\""
    ↪ )
        manifestFile.write(manifestContent)
        // Increment version code from version codes properties file
        def versionCodesFile = project.file(versionCodesFilePath)
        def versionPattern = Pattern.compile(applicationId + "=(\\d+)")
        def versionCodesText = versionCodesFile.getText()
        def versionMatcher = versionPattern.matcher(versionCodesText)
        versionMatcher.find()
        def newVersionCodesText = versionMatcher.replaceAll(applicationId + "=" +
    ↪ versionCode)
        versionCodesFile.write(newVersionCodesText)
      }
    }

    //Make release config generation depend on renaming
    project.tasks.whenTaskAdded { task ->
      if (task.name == 'generateReleaseBuildConfig') {
        task.dependsOn 'renameApk'
      }
    }

    // Set keystore path
    project.android {
      signingConfigs {
        release {
          Properties props = new Properties()
          if (project.rootProject.file(keyStorePath).exists())
          {
            props.load(new FileInputStream(project.rootProject.file(keyStorePath)))
            storeFile project.rootProject.file(props['storeFile'])
            storePassword props['storePassword']
            keyAlias props['keyAlias']
            keyPassword props['keyPassword']
```

```
92          }
93        }
94      }
95    }
96
97    project.android {
98      buildTypes {
99        release {
100         signingConfig signingConfigs.release
101         zipAlignEnabled = true
102       }
103     }
104   }
105
106   project.android.dexOptions.preDexLibraries = !project.getRootProject().hasProperty
      ↪ ('disablePreDex')
107 }
108
109 // Creates a new app in the version code file
110 def makeNewApp(manifestFile, versionCodesFile, manifestMatcher, applicationId) {
111   def versionCode = 1
112
113   // Write version code to manifest
114   def manifestContent = manifestMatcher.replaceAll("versionCode=\"" + versionCode +
      ↪ "\"")
115   manifestFile.write(manifestContent)
116
117   // Add version code to version codes properties file
118   def versionCodesText = versionCodesFile.getText()
119   def newVersionCodesText = versionCodesText + "\n" + applicationId + "=" +
      ↪ versionCode
120   versionCodesFile.write(newVersionCodesText)
121 }
122
123 def getVersionCode(project, applicationId) {
124   // Load version codes properties
125   if (project.rootProject.file(versionCodesFilePath).exists()) {
126     Properties versionsProp = new Properties()
127     versionsProp.load(new FileInputStream(project.rootProject.file(
      ↪ versionCodesFilePath)))
128     // If version code already exists
129     if (versionsProp[applicationId] == null) {
130       // no version code was found for this project, make a new one!
131       def versionCodesFile = project.file(versionCodesFilePath)
132       def manifestFile = project.file(manifestFilePath)
133       def pattern = Pattern.compile("versionCode=\"(\\d+)\"")
134       def manifestText = manifestFile.getText()
135       def matcher = pattern.matcher(manifestText)
136       makeNewApp(manifestFile, versionCodesFile, matcher, applicationId)
137       return [value: 1, created: true]
138     } else {
139       // Load version code. Increment as the incrementation task is run after thus
      ↪ task
140       return [value: Integer.parseInt(versionsProp[applicationId]), created: false]
```

```
      }
    }
    else {
      return [value: 1, created: true]
    }
  }
}

class GooglePlayAccount {
  String accountEmail = '845822318110-g2fur4jtaa6g1jek6n4m5i3jbpmfi6em@developer.
    ↪ gserviceaccount.com'
  String keyPath = '/var/lib/jenkins/google_play_keys/google_play_api_key.p12'
}
```

Listing B.1: Gradle Plugin to help publish APKs (written in Groovy)

```
package dk.giraf.gradle

import org.gradle.api.Plugin
import org.gradle.api.Project
import org.gradle.api.GradleException
import org.gradle.api.publish.maven.MavenPublication

class LibraryPlugin implements Plugin<Project> {

  def artifactUrlSnapshots = 'http://cs-cust06-int.cs.aau.dk/artifactory/libraries-
    ↪ snapshots'
  def artifactUrlReleases = 'http://cs-cust06-int.cs.aau.dk/artifactory/libraries-
    ↪ releases'
  def artifactCredentialsPath = '/srv/jenkins/credentials/artifactory.properties'

  void apply(Project project) {
    // Create giraf plugin extension
    GirafLibrary girafLib = project.extensions.create ('girafLibrary', GirafLibrary)

    // Set version name
    if(!project.hasProperty('versionName')) {
      project.ext.set('versionName', '0.0-SNAPSHOT')
    }
    // Apply plugins needed for libraries
    project.afterEvaluate {
      // Read artifactory credentials
      Properties props = new Properties()
      if (project.rootProject.file(artifactCredentialsPath).exists()) {
        props.load(new FileInputStream(project.rootProject.file(
    ↪ artifactCredentialsPath)))

        project.publishing {
          publications {
            repositories.maven {
              if(project.versionName.endsWith('-SNAPSHOT')) {
                url artifactUrlSnapshots
              } else {
                url artifactUrlReleases
```

```
36              }
37              credentials {
38                username props['username']
39                password props['password']
40              }
41            }
42          }
43        }
44      }
45      // Add publication even when no credentials file exists. It is used for local
    ↪ publishing.
46      project.publishing {
47        publications {
48          maven(MavenPublication) {
49            groupId 'dk.aau.cs.giraf'
50            artifactId project.girafLibrary.libraryName
51            version project.versionName
52            artifact 'build/outputs/aar/library.aar'
53          }
54        }
55      }
56      // Add rename dependency
57      project.tasks.findByName('generateReleaseBuildConfig').dependsOn(project.tasks.
    ↪ findByName('renameLib'))
58
59      project.tasks.findByName('publishToMavenLocal').dependsOn project.tasks.
    ↪ findByName('build')
60
61      project.android.dexOptions.preDexLibraries = !project.getRootProject().
    ↪ hasProperty('disablePreDex')
62    }
63
64    project.task('renameLib') << {
65      // Set library output name
66      project.android.libraryVariants.all { variant ->
67        variant.outputs.each { output ->
68          def outputFile = output.outputFile
69          if (outputFile != null && outputFile.name.endsWith('.aar')) {
70            def fileName = "library.aar"
71            output.outputFile = new File(outputFile.parent, fileName)
72          }
73        }
74      }
75    }
76  }
77 }
78
79 class GirafLibrary {
80   String libraryName
81 }
```

Listing B.2: Gradle Plugin to publish libraries (written in Groovy)

C Dependency Workflow Guide

Alle biblioteker bliver published på et Maven repository Artifactory (felter skal være tomme for at logge ind) når de bygges på Jenkins. Hvis man ønsker at udgive en version, skriver man (et vilkårligt sted) i sin git-besked `@minor`, `@major` eller `@patch`, for at udgive henholdsvis en minor, major og patch release. Ved en patch går for eksempel version `1.0.0` til `1.0.1`. Ved minor release går for eksempel version `1.1.2` til `1.2.0`, og ved en major release går for eksempel version `1.1.4` til `2.0.0`. Når der committes uden `@patch`, `@minor` eller `@major` bliver der udgivet et snapshot (udviklingsversion). Et snapshot har eksempelvis versionen `1.1.5-SNAPSHOT`, hvis den NÆSTE patch-release er `1.1.5`. Forskellen på patch, minor og major releases er beskrevet her. Kort fortalt:

- `MAJOR` angiver ændringer som ikke er bagud-kompatible.
- `MINOR` er ny funktionalitet med bagudkompatibilitet.
- `PATCH` er bagud-kompatible bug-fixes.

For at angive en dependency til et bibliotek, tilføjer man den i dependencies-sektionen i `build.gradle`-filen. For eksempel (Listing C.1):

```
1 dependencies {
2   compile ('com.android.support:support-v4:+')
3   compile(group: 'dk.aau.cs.giraf', name: 'girafComponent', version: '1.0.0', ext:
      ↪ 'aar')
4   compile(group: 'dk.aau.cs.giraf', name: 'oasisLib', version: '1.0.1', ext: 'aar')
5   compile(group: 'dk.aau.cs.giraf', name: 'localDb', version: '1.0.1', ext: 'aar')
6   compile(group: 'dk.aau.cs.giraf', name: 'meta-database', version: '1.0.4')
7 }
```

Listing C.1: Dekleration af dependencies

C.1 Lokal test

Hvis man vil teste et bibliotek lokalt uden at uploade det til Jenkins, kan man køre Gradle-task'en build efterfulgt af publishToMavenLocal, hvilket lægger en version `0.0-SNAPSHOT` op på et lokalt maven repository. Nu kan dette bibliotek tilgås ved version `0.0-SNAPSHOT`. For eksempel: `compile(group: 'dk.aau.cs.giraf', name: 'girafComponent', version: '0.0-SNAPSHOT', ext: 'aar')`

Der skal desuden tilføjes en reference til et lokalt maven repository (mavenLocal()), som vist her (Listing C.2):

```
1 repositories {
2     mavenCentral()
3     maven {
```

```
        url 'http://cs-cust06-int.cs.aau.dk/artifactory/libraries'
    }
    mavenLocal() // <-- HER
}
```

Listing C.2: Dekleration af repositories

For at sikre, at det nyeste `SNAPSHOT` altid bliver downloaded, kan følgende tilføjes til `build.gradle` i det projekt, som anvende biblioteker (det er måske allerede tilføjet) (Listing C.3):

```
// Always download latest snapshot
configurations.all {
    resolutionStrategy.cacheChangingModulesFor 0, 'seconds'
}
```

Listing C.3: Kode til at altid at bruge det nyeste snapshot

D Git Hooks

D.1 Post-receive hook

```
#!/usr/bin/python

import os
import subprocess
import re
import fileinput
import requests
from requests.exceptions import ConnectionError
from requests.exceptions import Timeout

VERSION_FILE = '/var/lib/jenkins/project_version_codes/libversion'
JENKINS_BASE_URL = 'http://localhost/jenkins'
GIT_RO_BASE_URL = 'http://cs-cust06-int.cs.aau.dk/git-ro'

class Library(object):
    def __init__(self, name, major, minor, patch, jobname):
        self.name = name
        self.major = major
        self.minor = minor
        self.patch = patch
        self.jobname = jobname

def main():
    """Main function for handling push"""
    # Load libraries
    libraries = load_libraries()
    # Read repo name
    repo = re.sub('.git', '', execute_sh_cmd('basename $(pwd)')[0])
    # If repo is not defined as a library, just trigger Jenkins the regular way.
    if not is_library(libraries, repo):
        try:
            trigger_jenkins_poll(repo)
            print "Thank you for your push. Jenkins will be serving you in a moment."
        except (ConnectionError, Timeout) as e:
            print "ERROR TRIGGERING BUILD"
            print e
        return

    # Read from stdin
    for line in fileinput.input():
        old_rev, new_rev, refs = line.split()
        # Only deploy if on master branch
        if not 'master' in refs:
            continue
        recent_msg = None
```

```
46          # Get commits in push
47          commit_ids = get_commit_ids(old_rev, new_rev)
48          for commit in commit_ids:
49              commit_msg, exit_code = execute_sh_cmd('git log --format=%%B -n 1 %s' %
    ↪ (commit,))
50              if commit == new_rev:
51                  if exit_code == 0:
52                      recent_msg = commit_msg
53                  continue
54              if exit_code == 0:
55                  trigger_build(commit_msg, commit, libraries, repo)
56
57          if recent_msg != None:
58              trigger_build(recent_msg, new_rev, libraries, repo, True)
59
60      write_libraries_to_disc(libraries)
61
62  def is_library(libraries, repo_name):
63      has_lib = False
64      for lib in libraries:
65          if repo_name == lib.name:
66              has_lib = True
67              break
68      return has_lib
69
70  def load_libraries():
71      """Loads all known libraries and their current versions from the property file."""
72      libraries = []
73      if os.path.isfile(VERSION_FILE):
74          with open(VERSION_FILE) as v:
75              # Regex for matching lib name, major version, minor version, and Jenkins
    ↪ job name
76              rg = re.compile('((?:\w*-*)+), major=(\d*), minor=(\d*), patch=(\d*),
    ↪ jobname=((?:\w*-*)+)', re.IGNORECASE|re.DOTALL)
77              libs = v.readlines()
78              for lib in libs:
79                  m = rg.search(lib)
80                  if m:
81                      libraries.append(Library(m.group(1), m.group(2), m.group(3),
    ↪ m.group(4), m.group(5)))
82              v.close()
83      return libraries
84
85  def write_libraries_to_disc(libraries):
86      """Writes the libraries back to the libary file"""
87      # Read updated versions
88      versionFile = open(VERSION_FILE, 'wr+')
89      versionFile.truncate()
90      for repo in libraries:
91          versionFile.write('repo=%s, major=%s, minor=%s, patch=%s, jobname=%s\n' %
    ↪ (repo.name, repo.major, repo.minor, repo.patch, repo.jobname))
92      versionFile.close()
93
94  def execute_sh_cmd(cmd):
```

```
95      """Executes the given command
96
97      Return value: Tuple containing (output, exit code).
98      """
99      p = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE)
100     val = p.stdout.read().rstrip('\n')
101     exit_code = p.wait()
102     return val, exit_code
103
104 def get_commit_ids(start_id, end_id):
105     """Returns commit ids from the given interval, from first to last"""
106     output, exit_code = execute_sh_cmd('git rev-list --reverse %s..%s' % (start_id,
        ↪ end_id))
107     if exit_code != 0 or output == "":
108         return []
109
110     # Parse output
111     return output.split('\n')
112
113
114 def print_version_name(version_name):
115     """Pretty-prints the version name."""
116     print "-------------"
117     print "RELEASE: %s" % version_name
118     print "-------------"
119
120 def trigger_jenkins_poll(repo_name):
121     """Sends a poll build request to Jenkins."""
122     url = "%s/git/notifyCommit" % (JENKINS_BASE_URL)
123     payload = {'url': "%s/%s" % (GIT_RO_BASE_URL, repo_name)}
124     requests.get(url, params=payload)
125
126 def send_build_request(job_name, version_name, commit_id):
127     """
128     Sends a build POST request to Jenkins.
129
130     Parameters:
131         job_name:     The name of the Jenkins job building this library.
132         version_name: The version name of the library.
133         commit_id:  The corresponding commit id.
134     """
135     url = "%s/view/Libraries/job/%s/buildWithParameters" % (JENKINS_BASE_URL,
        ↪ job_name)
136     payload = {'libVersion': version_name, 'commitId': commit_id}
137     requests.post(url, data=payload)
138
139 def trigger_build(commit_msg, commit_id, libraries, repo_name,
        ↪ publish_snapshot=False):
140     """
141     Triggers a build on Jenkins.
142
143     Parameters:
144         commit_msg:       The contents of the commit message. Used for versioning.
145         commit_id:        The id of the commit to build.
```

```
        libraries:        The known libraries and their versions.
        repo_name:        The name of the repository.
        publish_snapshot: If true, snapshots will be published.
    """
    version_name = ""
    for lib in libraries:
        if lib.name == repo_name and commit_msg != None:
            new_major = lib.major
            new_minor = lib.minor
            new_patch = lib.patch
            if "@major" in commit_msg.lower():
                new_major = str(int(lib.major) + 1)
                new_minor = "0"
                new_patch = "1"
                version_name = "%s.0.0" % (new_major,)
            elif "@minor" in commit_msg.lower():
                new_minor = str(int(lib.minor) + 1)
                version_name = "%s.%s.0" % (lib.major, new_minor)
                new_patch = "1"
            elif "@patch" in commit_msg.lower():
                version_name = "%s.%s.%s" % (lib.major, lib.minor, lib.patch)
                new_patch = str(int(lib.patch) + 1)
            else:
                if not publish_snapshot:
                    return
                version_name = "%s.%s.%s-SNAPSHOT" % (lib.major, lib.minor, lib.patch)
            # Send request
            try:
                send_build_request(lib.jobname, version_name, commit_id)
                # Apply lib changes (request succeeded)
                lib.major = new_major
                lib.minor = new_minor
                lib.patch = new_patch
                print_version_name(version_name)
            except (ConnectionError, Timeout) as e:
                print "ERROR TRIGGERING BUILD"
                print e
            return

if __name__ == "__main__":
    main()
```

Listing D.1: Post-receive git hook for managing library releases and triggering Jenkins (written in Python)

D.2 Pre-receive hook

```
#!/usr/bin/python
import fileinput
import subprocess
import os.path

```

```
6  GIT_DISABLED_FILE = "/var/lib/jenkins/git_disabled"
7
8  def main():
9      if not check_jenkins_enabled():
10         print "Jenkins is busy at the moment. Please try again soon."
11         exit(1)
12     # Read from stdin
13     for line in fileinput.input():
14         old_rev, new_rev, _ = line.split()
15         # Check if contains snapshot
16         check_snapshot(old_rev, new_rev)
17         # Check if contains dynamic version reference
18         check_dynamic_version(old_rev, new_rev)
19         # Check if DatabaseCredentials is added
20         check_database_credentials(old_rev, new_rev)
21
22 def check_snapshot(old_rev, new_rev):
23     """Checks if a snapshot has been added"""
24     _, exit_code = execute_sh_cmd('git diff %s %s | grep + | grep 0.0-SNAPSHOT' %
       ↪ (old_rev, new_rev))
25     if exit_code == 0:
26         print "Jenkins is dissapointed:"
27         print "It looks like you are referencing a 0.0-SNAPSHOT."
28         print "Please fix this and commit again."
29         print "\nCommit hash: %s" % (new_rev,)
30         exit(1)
31
32 def check_dynamic_version(old_rev, new_rev):
33     """Checks if a dynamic version has been added"""
34     _, exit_code = execute_sh_cmd('git diff %s %s -- \'*.gradle\' | grep + | grep
       ↪ "[[:digit:]]\.+"' % (old_rev, new_rev))
35     if exit_code == 0:
36         print "Jenkins is dissapointed:"
37         print "It looks like you are referencing a library using a dynamic version
       ↪ number (e.g. 2.2.+)"
38         print "Please fix this and commit again."
39         print "\nCommit hash: %s" % (new_rev,)
40         exit(1)
41
42 def check_database_credentials(old_rev, new_rev):
43     """Checks if database credentials file has been added"""
44     diff_files, _ = execute_sh_cmd('git diff --name-only %s %s' % (old_rev, new_rev))
45     for new_file in diff_files:
46         if 'dk/aau/cs/giraf/localdb/DatabaseCredentials.java' in new_file:
47             print "Jenkins is dissapointed:"
48             print "It looks like you have commited the DatabaseCredentials file."
49             print "Please remove the file, clear your history and commit again."
50             print "\nCommit hash: %s" % (new_rev,)
51             exit(1)
52
53 def check_jenkins_enabled():
54     """Checks if git is enabled"""
55     if not os.path.isfile(GIT_DISABLED_FILE):
56         return True
```

```
57      f = open(GIT_DISABLED_FILE, 'r')
58      content = f.read()
59      if "1" in content:
60          return False
61      else:
62          return True
63
64  def execute_sh_cmd(cmd):
65      """Executes the given command
66
67      Return value: Tuple containing (output, exit code).
68      """
69      p = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE)
70      val = p.stdout.read().rstrip('\n')
71      exit_code = p.wait()
72      return val, exit_code
73
74
75  if __name__ == "__main__":
76      main()
```

Listing D.2: Pre-receive git hook for checking for common mistakes (written in Python)

E Download and Install APKs Powershell Script

```
# Downloads and installs the newest APKs.
# Author: Group sw609f15

param(
    [string] $baseurl = 'ftp://cs-cust06-int.cs.aau.dk',
    [string] $remotedir = 'newest_releases',
    [string] $username = 'ftpuser',
    [string] $password = '9scWKbP',
    [switch] $verbose = $false
)

function DownloadFile($sourcepath, $targetpath) {
    # Setup FTP request
    $ftprequest = [System.Net.FtpWebRequest]::create($sourcepath)
    $ftprequest.Credentials = New-Object System.Net.NetworkCredential($username,
    ↪ $password)
    $ftprequest.Method = [System.Net.WebRequestMethods+Ftp]::DownloadFile
    $ftprequest.UseBinary = $true
    $ftprequest.KeepAlive = $false
    $ftprequest.Timeout = 5000
    $ftprequest.ReadWriteTimeout = 5000

    # Send the FTP request to the server
    $ftpresponse = $ftprequest.GetResponse()
    $responsestream = $ftpresponse.GetResponseStream()

    # Create download buffer and local file to download to
    try {
        $targetfile = New-Object IO.FileStream ($targetpath, 'Create')
        if ($verbose) { Write-Host "File created: $targetpath" }
        [byte[]]$readbuffer = New-Object byte[] 1024

        # Loop through the download stream and send the data to the target file
        do {
            $readlength = $responsestream.Read($readbuffer, 0, 1024)
            $targetfile.Write($readbuffer, 0, $readlength)
        }
        while ($readlength -ne 0)

        $targetfile.close()
    } catch {
        $_ | fl * -Force
    }

    $ftpresponse.Close()
```

```
}

function ListFiles($path) {
    # Setup FTP request
    $ftprequest = [System.Net.FtpWebRequest]::create($path)
    $ftprequest.Credentials = New-Object System.Net.NetworkCredential($username,
    ↪ $password)
    $ftprequest.Method = [Net.WebRequestMethods+Ftp]::ListDirectory
    $ftprequest.UseBinary = $true
    $ftprequest.KeepAlive = $false
    $ftprequest.Timeout = 5000
    $ftprequest.ReadWriteTimeout = 5000

    # Send the FTP request to the server
    $ftpresponse = $ftprequest.GetResponse()
    $responsestream = $ftpresponse.GetResponseStream()

    # Parse the FTP response
    $FTPReader = New-Object System.IO.Streamreader($ResponseStream)
    $list = @()
    while ($line = $FTPReader.ReadLine()) {
        $list += $line
    }

    $FTPReader.Close()
    $ftpresponse.Close()

    return $list
}

# List all files in the FTP directory
$remotepath = $baseurl + '/' + $remotedir
$list = ListFiles $remotepath

if ($verbose) {
    Write-Host '-----------------------------------------------------'
    Write-Host 'Found the following files on FTP:'
    Write-Host '-----------------------------------------------------'
    Foreach ($item in $list) { Write-Host $item }
}

# Find ADB executable
$adb = Get-Command adb -ErrorAction SilentlyContinue | Select-Object -ExpandProperty
    ↪ Definition -First 1
if (!$adb) {
    # ADB not in path. Try default installation path
    $appdata = Join-Path ${env:localappdata}
    ↪ Android/android-sdk/platform-tools/adb.exe
    $adb = Get-Command $appdata -ErrorAction SilentlyContinue | Select-Object
    ↪ -ExpandProperty Definition -First 1
}
if (!$adb) {
    Write-Host 'ERROR: Could not find adb executable. Please put it in PATH.'
}
```

```
95
96  # Download these files
97  $count = 0
98  Foreach ($item in $list) {
99      # Determine the remote path for this item
100     $remoteitem = $baseurl + '/' + $item
101
102     # Determine the local path for this item
103     $filename = Split-Path $remoteitem -leaf
104     $localitem = Join-Path $PSScriptRoot $filename
105
106     DownloadFile $remoteitem $localitem
107     $count += 1
108
109     # Uninstall old versions and install new version
110     if ($adb) {
111         $packagename = [io.path]::GetFileNameWithoutExtension($filename)
112         Write-Host "Trying to uninstall $packagename..."
113         & $adb 'uninstall' "$packagename"
114         Write-Host "Trying to install $filename..."
115         & $adb 'install' '-r' "$localitem"
116     }
117 }
118 Write-Host "Found, downloaded, and installed $count files."
```

Listing E.1: Powershell script that downloads and installs newest APKs

F Miscellaneous Server Scripts

These scripts are run as part of preparing for testing.

F.1 Installing all APKs Script

```
#!/bin/bash
install() {
    INSTALL_OUTPUT="$($ANDROID_HOME/platform-tools/adb -s $1 install $2)"
    echo $INSTALL_OUTPUT

    IS_SUCCESS="$(echo "$INSTALL_OUTPUT" | grep -i success)"

    if ! [ "$IS_SUCCESS" ]
    then
        echo "Error installing $2 on $1"
        exit 1
    fi
}
export -f install

# Exit if no APKs are given
if [ $# -eq 0 ]
then
    echo "No APKs found"
    exit 1
fi

SERIAL_NUMBER="$(/srv/scripts/get_serial_numbers.sh)"

for p in $@
do
    parallel install {1} ${p}\; exit $? ::: $SERIAL_NUMBER
done
```

Listing F.1: Bash script that install APKs on all connected devices in parallel

F.2 Starting Dummy Database Insertion Script

```
#!/bin/bash
dummy_inserter() {
    $ANDROID_HOME/platform-tools/adb -s $1 shell am start -n
    ↪ dk.aau.cs.giraf.dummydbinserter/dk.aau.cs.giraf.dummydbinserter.MainActivity
    sleep 1

```

```
6       IS_RUNNING="$($ANDROID_HOME/platform-tools/adb -s $1 shell ps | grep
        ↪ dk.aau.cs.giraf.dummydbinserter)"
7
8       while [ "$IS_RUNNING" ]
9       do
10          IS_RUNNING="$($ANDROID_HOME/platform-tools/adb -s $1 shell ps | grep
        ↪ dk.aau.cs.giraf.dummydbinserter)"
11          sleep 2
12      done
13
14      echo "finished dummy insertion on $1"
15  }
16  export -f dummy_inserter
17
18  SERIAL_NUMBER="$(/srv/scripts/get_serial_numbers.sh)"
19
20  parallel dummy_inserter ::: $SERIAL_NUMBER
```

Listing F.2: Bash script that starts and waits for the dummy database insertion app in parallel on all connected devices

G Continuous Integration Guideline

The objective of this guide is to describe how to use continuous integration (compared to for example feature branches) and why we do it in the Giraf project. First, we present a number of guidelines. These are followed by clarification and argumentation about why they are useful.

To do continuous integration, follow these guidelines:

- Integrate with the master branch at least daily.
- Write automated test for your code.
- It is OK to integrate features which do not yet work (but disable them with boolean flags or throw `NotImplementedException`).
- Everything on the master branch must compile and pass all tests.
- Test locally before integrating.
- Failing builds must be resolved as fast as possible.

G.1 Why Continuous Integration

There are several reasons that we use continuous integration:

To reduce large, error-prone merges The more frequent developers integrate with the master branch, the smaller merges are needed. Large merges can be confusing and prone to errors.

To find errors fast By integrating at least daily, incompatibilities will be found fast and new code is quickly tested. Code that is not working should be integrated as well, but disabled so that it is not executed. As such, Lint errors will still be detected and it will break if no longer compatible. Public methods which are not working should throw a `NotImplementedException` to ensure that it is not called from other parts before it works.

To avoid breaking builds before sprint review If features are developed on branches, they tend to be merged few days before the sprint review. Because the features are developed on independent branches, they may not be compatible.

To improve overall code quality Continuous integration automates and presents statistics about the code quality, such as test coverage, lint problems etc. This, however, requires that all code is on the master branch.

To automate the build pipeline For continuous integration to work, automation is required. Tests and code quality measures must be automatic.

To be able to provide the customer the latest version When using continuous integration, we are always certain that we have a stable version to give the customer. The master branch always results in a stable, tested, and working build.

H UI Test Guide

UI tests can test various UI actions and that those actions result in the correct result. For example, a test can click on a button and assert that the correct view opens.

An introduction on how to do UI testing can be found here[1].

As the introduction mentions, it is important to add the following to `build.gradle` of the app being tested:

```
android {
    defaultConfig {
        testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"
    }
    packagingOptions {
        exclude 'LICENSE.txt'
    }
}

dependencies {
    androidTestCompile 'com.android.support.test:testing-support-lib:0.1'
    androidTestCompile 'com.android.support.test.espresso:espresso-core:2.0'
}
```

Due to the various Giraf apps requiring extra information when starting and a database, some extra work has to be done. To demonstrate this, an example UI test for Launcher can be found in:

```
launcher/launcher-app/src/androidTest/java/dk/aau/cs/giraf/launcher/test/
    ↪ HomeActivityUITest.java
```

The test class extends the `ActivityInstrumentationTestCase2` class. The activity that one wishes to test must be in the <>. The following snippet tests the `HomeActivity`.

```
public class HomeActivityUITest
        extends ActivityInstrumentationTestCase2<HomeActivity>
```

The method `setUp` is run before each test. In this case it creates an intent with a guardian id to start the `HomeActivity` of the launcher. To do this a helper is created that creates some dummy data.

```
@Before
public void setUp() throws Exception {
    super.setUp();
    injectInstrumentation(InstrumentationRegistry.getInstrumentation());

    Intent intent = new Intent();
    intent.putExtra(Constants.GUARDIAN_ID, 1);
    setActivityIntent(intent);

```

[1] https://developer.android.com/training/testing/ui-testing/espresso-testing.html#setup

```
10      Helper h = new
        ↪ Helper(getInstrumentation().getTargetContext().getApplicationContext());
11      h.CreateDummyData();
12
13      mActivity = getActivity();
14  }
```

When each test has finished the database is cleared.

```
1   @After
2   public void tearDown() throws Exception {
3       super.tearDown();
4       Helper h = new
        ↪ Helper(getInstrumentation().getTargetContext().getApplicationContext());
5       h.clearTables();
6   }
```

The actual test itself clicks on the settings button on the home screen and checks that the window opens.

```
1   public void testSettingsButton() {
2       onView(withId(R.id.settings_button)).perform(click());
3       onView(withId(R.id.settingsListFragment)).check(ViewAssertions.matches(isDisplayed
        ↪ ()));
4   }
```

The test can then be run by running the `connectedCheck` target.

I Jenkins Structure

Here we describe the main points of interest of Jenkins.

I.1 Projects

Jenkins runs a number of jobs, or *projects*, that do various tasks. Most of these projects are *inheritance projects*, while some are *freestyle* projects. New projects can be created by the *New Item* button on the left.

For a project, mainly the following categories are configured:

Source Code Management What source code management system is used. For the multi-project Git is used. The repository to pull code from can be configured, as well as what branches to build.

Build Triggers When a build is triggered. Some projects are build periodically, like monkey tests, whereas apps and libraries are build whenever a push is made to their respective repositories.

The method for apps and libraries differ. Apps have the *Poll SCM* option ticked, which means that any project that has this option ticked will be run whenever a push is made to the repository defined in the project. The option will have a warning stating that no schedules will be run, but this is not correct. Libraries have no option in build triggers ticked, as the builds are started through the Git hook which sends a post request with build parameters.

Build Environment This is where the Android emulator is configured.

Build This category contains any actual build steps. This might include executing a Gradle script or executing shell.

Post-Build Actions Any tasks that must be done after the main build steps, such as publishing test reports or sending email notifications on build failure.

I.1.1 Abstract Projects

There are a number of abstract projects that are inherited to ease project configurations. Note that, of some unknown reason, not every setting can be inherited (so always verify it works if you add a setting in an abstract job). The abstracts projects are:

Android App Inheritable The main inheritable project for apps. Inherits a number of other projects to form the basis of apps. All app projects inherits from this project.

Android Lib Inheritable The same as the Android App Inheritable project, just for libraries.

gradle_lib_inheritable The base project for Gradle libraries.

Job failure email + log rotation Configures emails for job failure and discards old artifacts. Does not configure the emails for monkey tests.

monkey_test_baseline The base project for monkey tests. Configures the email for monkey tests.

Move APKs to ftp Moves release and debug APKs from a successful build to the ftp server.

Move Credentials The credentials for the local db are not managed by Git. This project moves the file stored on the server to the project workspace.

Publish APK Publishes release APKs of a successful build to Google Play.

Publish CodeCoverage Publishes code coverage results.

Publish Lib Uploads successfully built libraries to Artifactory.

Publish lint and test reports Publishes lint and test reports.

Run Checks Runs tests on an Android device.

Run emulator Configures the emulator that is run before each project is build.

I.1.2 Reports

When clicking on a project the lint and test results can be seen on the right. Clicking on a specific build will also give a code coverage reports.

I.1.3 Project Versions

The Jenkins plugin giving abstract projects also provides project versioning. We do not use this feature.

I.1.4 Shelved Projects

A project can be shelved meaning that it is disabled and zipped with its build history. When unshelving a project it is vital that there is no other project with the same name, as that project will be overwritten with no warning.

I.2 Managing Jenkins

In the *Manage Jenkins* page there are various managements options. At the top Jenkins will notify if there is an update available (which must be downloaded and installed manually) and if there are any major concerns.

Many items can be configured in the *configure system* page. Here the number of executers (number of concurrent builds) can be set. It is set to 1, as there can be issues if it is set to more than that. It is also here that email text is configured. The *default content* field looks as if it has a lot of random characters, but this is the Jenkins logo in ASCII art.

In the *configure global security* page the authorization of users can be managed. A user is automatically created when accessing Jenkins with one's student email as the user name. We advice to have a select few people have access to everything, and anonymous users to only have access to:

- Overall read
- Job read
- View read
- Job discover
- Job workspace

This way people cannot simply start project manually, which can interrupt continuous integration.

Finally Jenkins has a number of plugins to increase functionality. These can be managed in the *manage plugins* page. Be sure to regularly update plugins.

I.3 Jenkins Files on Server

Jenkins has a number of files on the server that are of interest. These can be seen in Figure I.1.

git_disabled If this file exists, all pushes to Git will be rejected. The text in the file will be printed to the user. This can be used to disable pushes while Jenkins is down for maintenance.

google_play_api_key.p12 The credentials for the Gradle Play Published plugin so that apps can be automatically uploaded to Google Play.

keystore.properties Contains credentials for signing APKs.

MainKeyStore.keystore The keystore file for signing APKs.

libversion Specifies the next version of each library. The version is automatically handled by a Git hook. Contains the repository to the library which must be manually added, as well as the project name in Jenkins (written jobname). If a project in Jenkins is renamed this must be changed. It is important to add a new entry if additional libraries are made, as it will otherwise not be built.

```
/srv/jenkins/
├── git_disabled
├── credentials/
│   ├── artifactory.properies
│   └── DatabaseCredentials.java
├── google_play_keys/
│   ├── google_play_api_key.p12
│   ├── keystore.properties
│   └── MainKeyStore.keystore
├── jobs/
│   ├── launcher/
│   │   ├── builds/
│   │   ├── workspace/
│   │   └── ...
│   └── ...
├── project_version_codes/
│   ├── libversion
│   └── versionCodes.properties
└── ...
```

Figure I.1: Jenkins file structure on server

versionCodes.properties Contains the next version code (not the version code shown to the user, but a strictly increasing integer corresponding to the `versionCode` property in the Android manifest file) for each Giraf app. This file is automatically handled.

artifactory.properties Contains the credentials for Jenkins to upload libraries to Artifactory.

DatabaseCredentials.java Contains the credentials for the local-db library. It is moved automatically whenever the local-db library is build in Jenkins.

In addition the `jobs/` directory contains each project in Jenkins. In Figure I.1 the launcher project is shown. Each project contains the latests builds etc. It also contains the `workspace` directory that has the source code etc. of the project.

J Wireless Router Setup

J.1 Login Information

Wireless SSID	`MariusTestNetworkTM`
Wireless Password	`SeGiraffen`
Management	`http://192.168.1.1`
	`http://172.25.11.91`
	`SSH root@192.168.1.1`
Username	`root`
Password	`routeradmin`

J.2 IPv4 WAN configuration

Applying the following IPv4 WAN settings will (in combination with the router's MAC address) put the router on a network visible to the server.

Name	Value
Type	Static
Address	`172.25.11.91`
Netmask	`255.255.255.0`
Gateway	`172.25.11.1`
DNS1	`172.18.21.2`
DNS2	`172.18.21.34`

However, a change in the network setup at Aalborg University is still needed. Contact Per Majdal, IT Services in Cluster 3. He can enable a specific LAN socket (for example in the group rooms) access to the server network. Bring him the MAC address of the router (important: See this in software, the sticker on the device has the *wrong* MAC address) and the LAN socket name (sticker next to the socket, e.g. `K6.41 F10`).

The scripts are located in `/www/cgi-bin/` and `/bin/`.

K Process Chapter from sw601f15

Authors: Martin Fruensgaard, Mads B. Nielsen, Tobias S. Jepsen, Lasse Jensen

0.1 Analysis of the GIRAF project process

During informal discussions within the group, both during the first and second sprint review, we agreed the process used for the Graphical Interface Resources for Autistic Folk (GIRAF) project is probably the cause for many of the issues the project suffers from. This section offers an analysis of the software development process of the GIRAF project, which can be used for process improvements on this or future semesters.

The software development process of the project is an agile process that heavily leans towards Scrum. Issues pointed out in this section are mistakes in the context of Scrum or the practices used that are likely the cause of some of the issues brought up at the sprint review and project meetings.

0.1.1 Dividing by responsibility, not functionality

Large projects require a scaling of the Product Owner (PO) role which is done by introducing a hierarchy of collaborating POs, shown on Figure 1[2][5]. Mike Cohn uses the phrase "Sharing Responsibility, Dividing Functionality" to describe the structure of the hierarchy. Dividing by functionality means that all teams work on some functionality that is delivered to end users. Figure 1 shows an example of such a division of an office suite. One group works on the functionality off unctions and another on the graphs of the Spreadsheet product line [2][5]. Alternatively, the division could be made by responsibility. Some teams would have the responsibility of the design of the graphical user interface, whereas others have responsibility for the component managing storage and opening of files. Yet another team may be responsible for testing the different components.

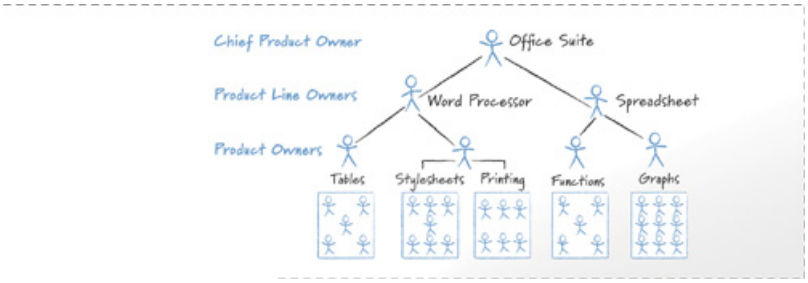


Figure 1: The project owner hierarchy on a large project[2].

In the GIRAF project division is made by responsibility. The Database (DB) subproject has responsibility for the OasisLib component for instance. Rather than viewing the project as a single product, the OasisLib component is instead seen as a separate product with the users primarily being the groups of the Graphical User Interface And Features (GUI) subproject.

The issue with this division of responsibility became apparent when formulating user stories.

The user stories handed to the DB subproject were tasks passed on to the DB subproject. Additionally, displaying value to our customers became almost completely trivial. Our customers, the teams working on other applications, would already have integrated our solution by the end of the sprint and thus had already seen the value of the solution.

This division has probably also existed in the previous semester, since it is unlikely that the developers of the GIRAF applications would have chosen the implementation discussed in Section **??**, because it makes querying the database quite cumbersome for the users of `OasisLib`. Additionally there has been a few complaints about using `OasisLib` for querying the database in reports from the previous semester, at the latest sprint planning and from group 12 with whom we are collaborating with during this sprint.

The obvious solution to the issue with dividing by responsibility is to make the division by functionality instead like the example shown on Figure 1.

0.1.2 No real Product Owner in the project

While it is generally a good idea for end users to be POs, because they are among the key stakeholders and as such have great knowledge of the problem domain. The PO role itself requires a high degree of availability, as well as attention to the interests of other stakeholders. The PO should also be a single person, to avoid conflicting messages, not a group ofi ndividuals[3][9].

The top level Product Owners (topPOs) show up for the sprint planning, prioritizing the product backlog by choosing which backlog items will be worked on in the current sprint. During the sprint itself, they are not available every day, although they can be contacted by mail. Additionally, there is another important stakeholder, namely the government, which has certain requirements about the protection of personal data in systems used in public institutions. This requirement represents a security concern that is often affected by the architecture of the system. The longer the requirement is ignored, the more cumbersome it will become to change the architecture. Furthermore, meeting this requirement is vital to the success of the GIRAF project, since it cannot be used in the public institutions of the topPOs, yet the topPOs have given it very low priority.

The responsibility of this potentially critical misprioritization does not rest on the topPOs alone, however, since the developers also have a responsibility to help the topPOs prioritize, though the topPO has the final word on the matter. Furthermore, the exact responsibilities of the topPO role have not have been presented and/or defined to the topPOs. Finally security concerns are non-functional requirements, which should not be captured by user stories and prioritized at all. Rather they should be listed as constraint cards or some other description of non-functional requirements[3]. It should be mentioned that security is just one of many alike constraints formulated as user stories.

Each subproject has their own sub-project Product Owner (subPO), however they let the users prioritize the product backlog and do not themselves understand the requirements

sufficiently for developers(other groups) to get elaborated details about them. Although user stories are reminder of a conversation, the product owners at different levels are either unavailable or does not understand the needs of theirs users in depth. In addition to this, user stories do not provide the necessary details to implement them. For these reasons, the use of user stories does not make much sense under the current project conditions[4][3]. This issue is also accentuated by the lack of acceptance criteria commonly associated with the use of user stories, and each group does not have a uniquely identifiable and dedicated PO[3][12] [2].

0.1.2.1 Possible solutions

A solution to this issue could be to make the topPOs aware that they are required to be much more available and to make subPO aware that they need to be able to answer all questions about the users needs. The topPOs are external customers and it may not be possible for them to be as available as they are required to fill the topPO role. The issue with a customer or user being unavailable for a majority of the time is however not new and a common solution is to use a user proxy, which is a person that is not a real customer or user, but functions as a PO or on-site customer for the project. The issue with this approach is that developers make poor user proxies[3].

Another issue regarding the use of user proxies that is more specific to the conditions of the GIRAF project is that except for the users and the semester coordinator, the entirety of the staff working on the project is replaced after the duration of a semester. This means that each year a new group or person needs to become a user proxy and spend a lot of time understanding the users' needs. A suggestion to solve this might be to spend some time documenting the needs of the users, however this defeats the purpose of writing user stories in the first place - to avoid detailed documentation of requirements[3].

User stories are not however the only way to document functional requirements in an iterative and incremental development approach. In Unified Process, a use case model is used[7] and in other agile approaches, including Scrum, use cases can be used as well. The use cases in this situation has the advantage that they are much more detailed and are intended to be part of more elaborate documentation. Just like user stories they evolve over time, however they cover more functional requirements than user stories, which may pose challenges when dividing work. Additionally, user stories are easily understood by users or POs whereas use cases are more technical and possibly harder to understand[4][11][3].

0.1.3 The product is seeing very limited use

One of the tenets of agile development is to deliver working software at the end of every sprint, which means a potentially shippable product increment[3][6].

The project has been in development since 2011, yet it is not used regularly in the institutions according to key stakeholder Mette Als, one of the users. While only a single semester is

spent on development each year, the core features described at the beginning of the project by Mette appear to be quite simple, but none of them have been entirely finished. These simple features were the ability to find and display pictograms, displaying sequences of pictograms to illustrate steps of an activity and a week schedule to reduce the overhead of changing their current physical week schedules.

The apps `WeekSchedule` and `Sequence` which are the apps implementing these features are however still being developed during the current sprint. Several lesser essential apps, such as the game apps, have been developed even though the more critical apps still lack essential features such as the ability to save the schedule.

This issue is in part due to the lack of PO involvement described in subsection 0.1.2. With only user stories and no PO to go to for a more detailed description of the story, some requirements from the PO are bound to be missed or overlooked. Additionally, the students developing the software project are not used to work together and show a preference to work on their own individual modules. This preference has likely lead to some of the features, which were never requested by the customer, such as the use of QR-codes, and to the incompleteness of the core functionality(By grabbing lesser important backlog items to work on).

Additionally the security concerns regarding the Danish data protection legislation have yet to be resolved. This means that the GIRAF project cannot be integrated into the institution.

While the non-functional requirements should have been built into the system from the beginning, the next best solution is to ensure it as soon as possible. Additionally, defining a minimum viable product (MVP) could be of value for the project. A MVP is the product that has exactly those features that allows it to be deployed. This is not just a potentially releasable product, but a product that is actually intended to be released to at least a subset of possible customers or users[13][1]. Once a MVP is defined, it should be implemented and deployed as fast as possible to the environment in which it will be used. An example of a MVP for a web shop could be a website where the user can browse items by title and buy each item individually.

0.1.4 Only features and bugs on the product backlog

User stories or use cases describing features or bugs are likely to be the most numerous type of backlog items appearing on a Scrum product backlog. There are however other items, such as technical work and knowledge acquisition items as well. Technical work items could be configuring Jenkins for continuous integration, setting up a test environment or major refactoring of some part of the system. Knowledge acquisition could be researching different libraries or technical solutions. These items may not directly provide business value, but can be necessary tasks. The PO and developer team should collaborate on prioritizing these items. The developer team should explain the value to the PO, and the PO must choose when it is convenient to perform these tasks as with any other backlog item[10][8].

An example of this could for instance be by swapping out the database system to enhance

portability. The PO could choose to postpone the technical work in favor of getting more total amount of user stories completed during the upcoming sprint.

In GIRAF the product backlog consists only of user stories and related tasks. The Build And Deployment (B&D) subproject is assigned technical work exclusively, such as setting up Jenkins. Rather than treat these stories as technical work to be prioritized by the PO, the B&D subproject is treated as a project making a product of their own with complimentary user stories. This view is identical to that of the DB subproject as discussed in subsection 0.1.1 and causes an over-emphasis on the type of technical work handled by the B&D sub-project. The GUI and DB sub-projects implement features and tasks, deemphasizing or ignoring potentially very valuable technical work or knowledge acquisition. In any case, the decision on whether or not to complete features or technical work is taken away from the PO.

Combining the division off unctionality described in subsection 0.1.1 with the addition of technical work and knowledge acquisition items on the product backlog will give back the decision making power to the PO.

0.1.5 Summary

The project is currently divided by responsibility, rather than functionality. This leads to artificial user stories from the different subproject groups to each other as if they were building separate products. The project should ideally be redivided by functionality not responsibility.

Another issue the project suffers from is the lack of a proper POs. The POs of the GIRAF project as a whole are not sufficiently available to fill the PO role. The subproject POs are available, but do not possess the necessary domain knowledge to explain or prioritize user stories and thus cannot fill the PO role. The user stories additionally lack acceptance criteria accentuating the issue. The user stories do not themselves describe the necessary detail to implement the feature they describe and having POs available to supply this detail is vital.

Solutions to this issue includes establishing a user proxy, however the obtained knowledge would be lost from semester to semester unless documented. Another solution is to stop using user stories and switch to use cases, which can be used for agile development as well. Use cases are generally larger than user stories and it may therefore be harder to divide work and may not be as easily understood by POs or users.

GIRAF is seeing very limited use. This is in part due to the lack of compliance to the Danish data protection legislation and part due to core features not being fully completed. The cause of this incompleteness likely stems from the lack of true POs and the students developing the system. The students are used to working in separate groups and not collaborate with other groups and thus show a preference to work on separate modules, perhaps artificially creating work. A definition of a MVP for the GIRAF project could give the project direction and provide valuable experience from the deployment of the system.

Only user stories and related tasks appear on the GIRAF backlog, which de-emphasizes the value of technical work and knowledge acquisition for the GUI and DB subprojects. Meanwhile the B&D subproject does nothing but technical work, perhaps over-emphasizing the importance of the type of technical work they are responsible for. Through all of this the PO is left out of the decision making.

NOTE: This bibliography is part of Appendix K. The bibliography of this report can be seen on page 121.

Bibliography

[1] Vladimir Blagojevic. The Ultimate Guide to Minimum Viable Products. Website, 2013. http://scalemybusiness.com/the-ultimate-guide-to-minimum-viable-products/.

[2] M. Cohn. *Succeeding with Agile: Software Development Using Scrum*. Pearson Education, 2009.

[3] Mike Cohn. *User stories applied: For agile software development*. Addison-Wesley Professional, 2004.

[4] Courtney. Use cases vs user stories in agile development. Website, January 2012. http://www.boost.co.nz/blog/2012/01/use-cases-or-user-stories/.

[5] Colin Bird & Rachel Davies. Scaling scrum. Presentation, 2007. https://www.scrumalliance.org/resource_download/287.

[6] Kent Beck et. al. Manifesto for agile software development. Website, February 2001. http://agilemanifesto.org/.

[7] Craig Larman. *Agile Iterative Development: A Manager's Guide*. Addison-Wesley, 2004.

[8] Ben Linders. Should You Create User Stories for Technical Debt? Website, March 2013. http://www.infoq.com/news/2013/03/user-stories-technical-debt.

[9] Scrum Methodology. Scrum Product Owner. Website. http://scrummethodology.com/scrum-product-owner/.

[10] Mountain Goat Software. Scrum Product Backlog. Website. http://www.mountaingoatsoftware.com/agile/scrum/product-backlog.

[11] Matt Terski. Agile Use Cases in Four Step. Website, September 2009. http://blog.casecomplete.com/post/Agile-Use-Cases-in-Four-Steps.

[12] Don Wells. Acceptance Tests. Website. http://www.extremeprogramming.org/rules/functionaltests.html.

[13] Wikipedia. Minimum viable product. http://en.wikipedia.org/wiki/Minimum_viable_product.

This bibliography is part of Appendix K