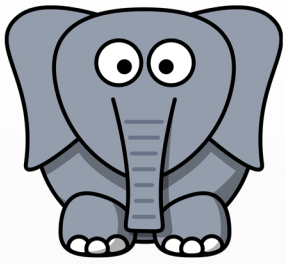
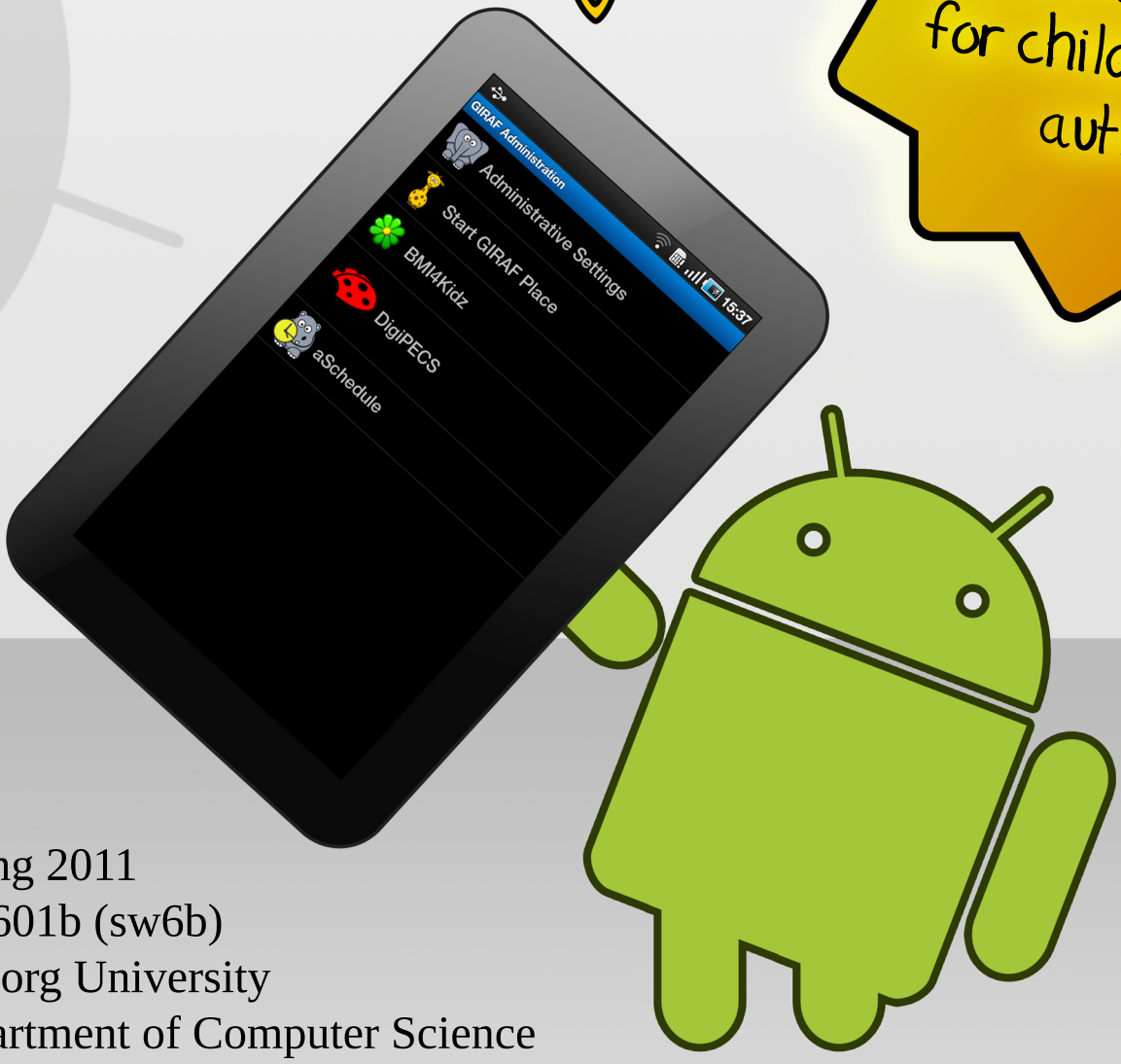


Administration Module for GIRAF



Spring 2011
f11s601b (sw6b)
Aalborg University
Department of Computer Science

Title:

Administration Module for GIRAF

Topic:

Application Development

Semester:

SW6, Spring 2011

Group:

f11s601b (sw6b)

Students:

Jacob Bang

Lasse Linnerup Christiansen

Steffan Bo Pallesen

Supervisor:

Ulrik Nyman

Copies: 5
Pages: 106
Published: May 27th, 2011

Aalborg University
Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg Øst
http://www.cs.aau.dk

Synopsis:

This report documents the process of developing the administration module for an Android software system called GIRAF. GIRAF is a software platform for the Android operating system that targets small children and children with mental disabilities. The purpose of the system is to mask the original Android functionality, and present the child with child friendly software that will aid the child in the everyday life. The software that GIRAF presents to the child, is an application launcher and a set of applications that guardians can download via the GIRAF market place. In order to configure these applications, an administration module targeted at guardians has been developed.

The GIRAF system has been developed by four independent development groups, where each group has been assigned to develop a sub-part of the system. This report focuses on the administration module sub-part, which has been developed by group sw6b, using a custom made development method. The administration module provides a service for GIRAF applications, allowing them to define administrative settings to be handled by the administration module. These settings can have different data types and constraints, which will all be defined in an XML file that must be included in the application. The values of these settings can be updated and retrieved by applications using a thoroughly documented library for the administration module which also has been developed this semester. The administration module also provides a Graphical User Interface (GUI) that can be used by guardians to edit the value of application's administrative settings. In order to ensure the stability of all these features, most parts of the administration module have been tested using unit-testing, regression-testing, and integration testing.

Preface

This report documents the 6th semester Software Engineering bachelor project, created by group sw6b from Department of Computer Science at Aalborg University. The project has been worked on during the period February 1st 2011 to May 27th 2011. The report covers the development of an administration module for a larger software system called GIRAF. GIRAF is a single user, module based overlay for the Android Operating System (OS) that targets small children and children with mental disabilities. The idea of GIRAF is to provide aiding facilities to the child, and help its parents and kindergarten teachers configuring the system. In general, the structure of the GIRAF system is based on the idea of separating and adjusting functionality. For the child using the system, only applications meeting the child's abilities should be available. And for the guardians responsible for administrating the system, all types of system and application settings should be available. Applying this separation helps the guardians setting boundaries and adapting the system to the child.

GIRAF is developed by different independent groups as a part of this semester's multi project. Two groups have been responsible for developing applications; one group has been responsible for developing a home screen and a market place, while our group has been responsible for implementing the administration module of GIRAF. Besides covering details about the multi project, GIRAF, and especially its administration module, this report also elaborates on the cooperation between the different groups during this multi project.

Originally, the "GIRAF" name was intended to be a code name for the multi project. However, no new name has ever been given to the project, which is why the code name is now the official name of the multi project. We never agreed upon what "GIRAF " is an abbreviation for, but we suggest a title like: "Graphical Interface Resources for Assistive Functionality".

Goal of the Semester

The topic of this semester is application development, and the goal is for the students to display the wide array of skills they have acquired during the previous five semesters. Furthermore, the students must also show expertise in the field of software testing, development, and project management [DI09, P 16].

Report Structure

The report is divided into four parts.

Part I, Multi project
In the first part, Chapter 1 introduces the problem that the multi project aims to solve. Chapter 2 describes the system definition and the overall architecture of GIRAF. Details on how multi project management has been handled are given in Chapter 3. In Chapter 4, a description of an integration test executed in order to ensure compatibility between the different modules of GIRAF is given. Finally, in Chapter 5 the first part is completed with a description of the technical details of the development- and software platform that GIRAF is developed for.

Part II, Administration Module
The second part of the report describes the development of the administration module for GIRAF. First, the development method used is described in Chapter 6. Then, in Chapter 7 the system architecture of the administration module is described in detail. In Chapter 8, a demo application named BMI4Kidz is used to describe how an application is integrated into the administration module of the GIRAF system. The technical details behind the implementation of the administration module, covering both its library and its engine, will be given in Chapter 9 and Chapter 10 respectively. In Chapter 11, the user interface of the administration module is described. This is the interface that the parents and kindergarten teachers will be supposed to use. Finally, in Chapter 12, test approaches used in our development, like unit testing, regression testing, and automated integration testing are described. Furthermore, Chapter 12 also gives an introduction to code coverage analysis in Android, and describes how that has been used in our project.

Part III, Recapitulation
Part three recapitulates on the entire project. In Chapter 13 the process of using our custom development method is evaluated. Future work on the administration module is covered in Chapter 14. Here we point out features that can be optimized or added. The idea is to keep the content at level, enabling future multi projects to understand the descriptions, and encourage them to improve the administration module even more. A conclusion on our sub-part of the multi project as well as a brief conclusion on the multi project is given in Chapter 15. Finally, in Chapter 16 we reflect on the collaboration between the development teams during the multi project. The goal of this chapter is to share some of our experiences and help improving the structure of future multi project semesters.

Part IV, Appendix
Part four lists the appendices of the report. Among different appendices, a list of acronyms is given in Appendix A and a list of contents on the enclosed DVD is specified in Appendix D.

Code License

It should be noted that the GIRAF project is developed as an open source project. The code is open for anyone and must always be open to ensure continuous contribution to the community involved in the development of GIRAF. The source code, as well as information about the GIRAF project can be found at its project site, located here: `http://code.google.com/p/sw6android/`.

Conventions

As a precondition for reading this report, the reader is expected to have knowledge about Java programming.

References are denoted in square brackets containing a part of the author's surname, and the year of publication. In other words, the reference [Pat06] refers to the book "Software Testing" written by Ron Patton, published in 2006. All references are collected in the bibliography list of this report.

Acronyms are used to abbreviate phrases and names. The first time an acronym is used, the word being abbreviated is written in its entire length followed by the acronym in parentheses. All acronyms are listed in Appendix A.

In code-listings, the ↵ symbol indicates a word-wrap. Furthermore, '. . .' indicates omitted code that is not relevant in the current context. Presented code samples are not expected to compile out of context.

The enclosed DVD contains material like the source code of GIRAF and the administration module, as well as an instance of this report in PDF-format. Websites from the web-references in the bibliography are located on the disc too. In Appendix D a complete list of the content of the enclosed DVD is given.

We would like to thank supervisor Ulrik Nyman from Department of Computer Science at Aalborg University for contributing to the project with constructive feedback.

Contents

Part I

Multi Project

In Chapter 1, this part starts by analyzing the problem that this multi project aims to solve. Developing a large system across multiple groups is a challenging task as many people are involved. Therefore, a structure for inter-group communication and project sharing has been setup at the start of the development process and is covered in Chapter 3. In order to get a clear picture of the system specification agreed upon by the groups, a FACTORS analysis and system definition has been created and is described in Section 2.1 and Section 2.2. Furthermore, an architectural plan for the entire software system has been created by the groups in order to give a technical overview. The overview can be seen in Section 2.3. This serves to help not only the groups, but also future developers who might want to extend the software system. Chapter 4 describes a multi project test that has been conducted to ensure that the modules developed by the different groups were compatible. Furthermore, the chapter discusses the results of this test. Finally, this part gives a technical description of the Android platform in Chapter 5.

1 Problem Analysis

The goal of this chapter is to define the problem, target group, as well as the platform for the multi project. All three areas have been discussed at group leader meetings where a representative from each group of the multi project has participated. As a common basis for these meetings, input from the project proposal, as well as from the multi project coordinator, the project related course in Multi-Project Management (Multi-Projekt Ledelse (MPL)), the student groups as well as their supervisors, were taken into account. This chapter will be based on the decisions made during these group leader meetings, trying to define the problem to be solved in the multi project, as well as the intended target group and target platform.

1.1 Problems

Children born with developmental disorders that cause severe mental handicaps can have difficulty communicating with others and/or structure their everyday life. One of these disorders is autism which exists in various degrees. The disorder affects a child's development of language and social skills, causing the autistic child to have a hard time feeling empathy towards others. It is normal that those children play by themselves escaping into their own world. Different facilities exist to ease the life of these children, and one of them is to give the children a more structured daily life. To help structure the daily life, a calendar is typically used. To aid communication, figures and pictures are often taken into use.

Because this multi project is focused on software, the *first problem* will be to investigate how it is possible to create a common software platform that can be used to practically aid the children in their daily life, and at the same time be used as a toy providing some kind of entertainment. It is reasonable to focus on the possibility of digitizing and deploying the facilities that already exists and are used, to a portable device. It is expected that the device is a smartphone or tablet, and that the device is to be used by a single child. As there exist different levels of mental disorders, it is expected that the platform can be configured and customized to each individual child.

Several of the facilities provided by this software platform will need to be configured on an ongoing basis. It could be done by adding new figures in the communication tool, or adding new activities to the calendar. Guardians (parents and kindergarten teachers) will take the role of performing these configurations. The *second problem* of this project will therefore be to determine how to design a software platform such that it can be configurable by a child's guardians. A part of this problem will be to make it simple for the guardians to configure the system, but at the same time not allow the child to gain access to the part of the system handling the configurations. It is also necessary that the child cannot bring the software into a state, where the child is locked and will have difficulties navigating away from.

Since the system will be configured on an ongoing basis to fit the child's needs, it should also be possible to add new or change functionality of the system. From a software perspective it is realistic to imagine that there continuously will be a development of facilities and new requirements for the system. Likewise, it will also be realistic that other third party companies wishes to integrate with the system and be a part of it. The *third problem* will be to design the software platform so there is support for the ongoing development

of the system and its software components. There should also be added an opportunity to easily add new software components to the system while still keeping it easy for the guardians to configure the system and its current facilities. Guardians should also be able to easily add and remove specific software components so the child only has access to specific functionality.

The *fourth and final problem* to consider in this multi project is how to make the system available on a hardware platform, that cannot easily be broken by small children. Modern smartphones and tables might be durable enough to be used for everyday use by adults and teenagers. However, with children and especially small children, the hardware might be exposed to serious physical challenges. As this multi project will only concern the software part of the system, this will not be a part of the project's scope.

To recapitulate the problems of this multi project, it focuses on how to develop a common software platform for a smartphone or tablet device that aids children with mental disorders by providing aiding facilities and some sort of entertainment. The software platform should be designed such that guardians can configure each system to meet the needs of the child using it. Also, the system should allow controlling the accessibility of different functionality on the device, and furthermore, it should be possible to extend the platform with new functionality, by installing software components. The platform should support ongoing development of the system and its software components. There will be no focus on the durability of the hardware devices, as this will be out of scope for this multi project.

1.2 Target Group

As the system is divided into a child-state and a guardian-state, it is reasonable to divide the target group into two different groups. The first group being the children and the second group being their guardians. The user-state is defined to be the part of the system which provides aiding facilities and entertaining applications. The target group for this part of the system will be the children. No age of these children is defined, as the project will be working with children having mental disorders, and in such cases you cannot equate age and level of mental disorder. It will therefore be likely that even older children will find the system useful. During development of the system, it is important to keep in mind that a simple and intuitive system with a minimal amount of states should be made. This requirement is not only relevant for the children with mental disorders, but for children in general, as a complicated system might easily cause confusion and/or frustration while attempting to use the system. Likewise, it should be taken into account that a child easily should be able to return to the home screen of the system, so the child does not experience getting lost in the system.

The second part of the target group is the guardians. As mentioned briefly in Section 1.1, guardians are defined to be the parents of the child as well as the child's kindergarten teachers. Both types of guardians can have different reasons to use the administrative part of the system; and this must be supported. Guardians may configure what is already installed on the system - for instance by adding new plans in a calendar, telling the child when different tasks are to be done, etc. Beyond using already installed functionality, the guardians might also want to add new content to the system, like new pictures, or they might want to add new applications or adjust the accessibility of already installed functionality. As both the parents as well as the kindergarten teachers are considered to have an average knowledge of how to use information systems, the system must meet the skills of this target group. If the system is going to be taken into use after this semester, it might even be a good idea for any company, providing the system, to arrange workshops where both parents and kindergarten teachers can be instructed in how to use it.

1.3 Target Platform

This section describes the target platform of the multi project. The platform will be described with regards to software and hardware.

1.3.1 Software

In the project proposal, it was suggested that Android was to be used as the software platform for the multi project, and based on that suggestion, Android has been chosen as the software-platform for the multi project. However, there exists many different versions of Android and in this section, we discuss which version the multi project should be based on. For a detailed description of the Android platform used in this project, see Chapter 5. The statistics and technical information written about Android in this section, is based on [And11i] unless otherwise stated.

Android is an OS for mobile phones and tablets, and is developed by Google Inc. The first version of the Android Software Development Kit (SDK) was released late September 2008 as version 1.0 [And08], and since that time, there have been a number of releases, version 3.0 being the latest. Today, version 1.6, 2.1, 2.2, and 2.3.3 are the most used versions of Android, with version 1.6 gradually being overtaken by newer versions. In regards to backward compatible concerns, Android is designed so that it is fully compatible with previous versions of the OS. For instance, this makes it possible for applications developed for 1.6 to be executed on an Android device running 2.3.3.

To keep Android devices up to date, some of the manufactures release updates to Android devices that was released with an older version of Android. This entails that older versions of Android quickly can lose market shares, as they are replaced by newer versions of the platform. Google is continuously making statistics showing which Android versions are visiting their own application market "Android Market" over a specified time period. Figure 1.1 shows the distribution of Android versions visiting Android Market the last 14 days of March 2011.

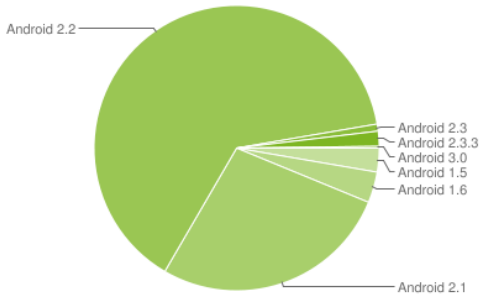


Figure 1.1. Shows the distribution of Android versions that have visited Android Market the last 14 days of March 2011.

As it can be seen from Figure 1.1, Android 2.1 with 27.2% market shares as well as Android 2.2 with 63.9% market shares are the most used Android versions these days. Naturally, this means that it will be these two versions which will candidate as software platform for the multi project. The advantage by choosing Android 2.1 over 2.2 is the high compatibility with already existing hardware. On the contrary it is most likely that new purchased hardware will be running Android 2.2 or higher as newer hardware is not shipped with Android 2.1.

For the multi project it has been decided to use Android 2.2. The decision is based on the statistics covered above, as well as the fact that newer hardware manufactured today is being shipped with 2.2 or newer.

1.3.2 Hardware

The project proposal does not define any requirements with regards to which specific hardware that should be supported by the multi project. Instead it is specified that the chosen hardware should support the chosen software platform and provide a touch-sensitive screen. The university has made different smartphones as well as a couple of tablets available to be used in the multi project. The following devices have been made available.

Phones:

- HTC Desire.
- HTC Hero.
- HTC Wildfire.

Tablets:

- Samsung Galaxy Tab.

The HTC Desire and the Samsung Galaxy Tab both meets the requirements defined by the project proposal as well as by the multi project groups, as both Android 2.2 and touch-sensitive screens are supported. However, the HTC Wildfire and the HTC Hero comes with Android 2.1, but can un-officially be upgraded to Android 2.2. Because these phones were designed when Android 2.1 was the newest version of Android, they might not perform well as other devices from a newer generation designed natively for Android 2.2. In addition to the requirements specified, the devices also have different functionality in common, including:

- Gyroscope for measuring phone orientation.
- Support for wireless network.
- Support for Bluetooth™ connections.
- Digital camera.
- USB connection.
- External memory.

During development, the different hardware platforms made available by the university will be used to test and run the system, and hence minimize the risk for developing product-specific functionality. As the final system is supposed to run on some sort of special designed hardware, made to resists any physical challenges a child might expose it to, it is likely that a mobile manufacturer is hired to design such a device. By ensuring that the system can be executed on a wide range of smartphones and tablets, it is expected that any compatibility issues are reduced as long as the device supports the functionality that is common to the hardware units used this semester. Besides this being good for any company that might use the multi project, this also adds an opportunity for guardians to test the system on a private Android device before buying a customized device to their child.

System Architecture 2

Based on the problem analysis in Chapter 1 and the problems covered in Section 1.1, this chapter covers the final system definition as well as the system architecture of GIRAF. It should be noted that the outcome of this chapter is solely based on cooperation between the multi project groups which has been administrated by the four group leaders elected at the beginning of the multi project. First, in Section 2.1 the basis of the system definition, the FACTORS criteria, is covered. Secondly, in Section 2.2 the system definition for GIRAF is specified, and finally, in Section 2.3, the architecture of GIRAF is defined.

2.1 FACTORS

The FACTORS criteria are used to support the preparation of a system definition. In the following, the FACTORS for GIRAF will be covered. The definition of each criterion is based on [MMMNS01].

Functionality *Describes the systems functions that support the application domain tasks. That is, defining what the system is able to do.*
The system should offer installation of new applications and make it possible to administrate common settings by need. The system should mask the normal functionalities of the unit to the user. Furthermore, the system should give the opportunity to control the usage of and access to system- and user profile settings as well as applications according to the current time, and location of the unit. The system should be delivered with a number of pre-installed applications which is customizable to the user.

Application Domain *Concerns those parts of an organization which administrate, monitor, or control a problem domain.*
Children with limited mental capabilities due to handicap or age, making it hard for them to handle the complexity of a normal smart-phone or tablet OS. Parents and kindergarten teachers (guardians) will be in charge of administrating the system.

Conditions *Covers conditions under which the system will be developed and used.*
The project is being developed by a number of study groups as a study project, and thus has a hard deadline that cannot be exceeded. The system should be simple and intuitive to use. The system should be developed such that it is customizable to the individual child and its disabilities. Furthermore, it should allow guardians to limit the functionality of the system. To allow other application developers to continue to develop the system and further applications after this semester, the system should be maintainable.

Technology *Covers the technology used to develop the system and the technology on which the system will run.*
The system must run on Android touch devices such as smart-phones and tablets. Different hardware should be supported, although it is required that the unit is running Android 2.2 or newer. The system should mainly be developed using Java and the Android SDK version 8 for Android 2.2.

Objects *Describes the main objects in the problem domain.*
A smart-phone or tablet device. The Android platform. Global system- and application specific settings. Applications.

Responsibility *Covers the systems overall responsibility in relation to its context. That is, how the system would interact with the tasks to be solved using the system.*
The system should act as an assistive tool by providing pre-installed applications developed to aid and entertain the small-aged and disabled children using the system. Furthermore, the system should provide the opportunity to install other third party applications. Through a home menu, the system should in accordance with the location, the user profile as well as the global settings of the system control which applications the user is allowed to access.

Sub Systems *The subsystems in the overall system.*
An administration module should provide access to user profile properties as well as global and specific application and system settings. A home menu must provide access to applications in accordance with the location, the user profile and the global settings of the system. Pre-installed applications including a day-planning tool and a Picture Exchange Communication System (PECS) application.

2.2 System Definition

In this section, the system definition of GIRAF is given:

A simple and intuitive module based single user system for Android touch devices, such as smart-phones and tablets. By masking the normal interface of an Android device, the device should offer functionality that is suitable for the intended user.

The system should be responsible for aiding and entertaining children with limited mental capabilities due to mental handicap and/or age, having a difficult time handling the complexity of a normal smart-phone or tablet OS. Guardians should be able to administrate the system by controlling selected application-, system- and user-specific settings through an administration interface on the phone.

Based on these settings, as well as the location of the unit, a home menu should be responsible for providing access to applications that conforms to the current settings and the state of the system. It should be possible for any third party to develop and provide additional applications to the system.

Beyond that, the system must be delivered with a set of pre-installed applications consisting of a visual, day-to-day, planning tool, and a PECS application. The system should be developed using Java and the recent version of the Android SDK. It is expected that the system supports Android 2.2. Furthermore, it is expected that the system is maintainable to such a degree, that it allows other developers to keep developing the system as well as applications to the system after this semester.

2.3 Components

In this section, the components of the multi project and their relations will be described. The component design is the outcome of work produced by a workgroup formed with the single purpose of defining the component architecture of the entire multi project.

The final component architecture that every group of the multi project has agreed upon, is seen in Figure 2.1.

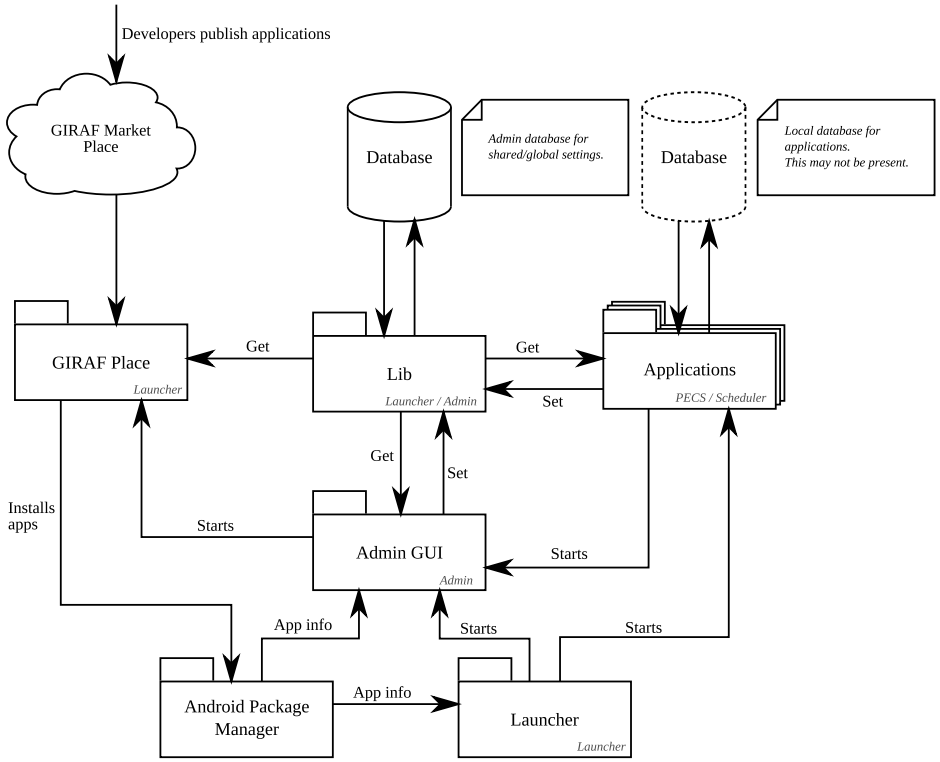


Figure 2.1. Components of the multi project.

In the top left corner of Figure 2.1, the market place of the multi project is seen. The market-place is connected to the Internet, and allows developers to add applications that should be available to all Android devices running the GIRAF system. *GIRAF Place* is a client for the application market, and it enables the guardians to download, install, and uninstall applications. From the *Lib*, various information about the user of the system can be retrieved. *GIRAF Place* uses this information to filter which applications should be available for download. In this way, only applications suitable for the user of the GIRAF system will be available. When an application has been downloaded through *GIRAF Place*, it will be installed by the Android Package Manager. In this process, the Android Package Manager first installs the package, and afterwards, tells the administrative part of the system, that it may now take care of storing any administrative settings the application might have. After installation, the installed application will be available in the *Admin GUI*, where all of its administrative settings can be adjusted. For security reasons, only guardians are intended to have access to the administrative part of the system. Furthermore, if the installed application meets the properties of the user, the application can be started from the *Launcher*.

Any application in the GIRAF system has the possibility to make use of the *Lib*. Besides various user information, the *Lib* also provides an opportunity for the applications to represent settings that should be editable when the guardians has brought the system into an administrative mode. The *Lib* has its own database for persistent data storage, where user information as well as administrative application-specific settings are stored.

In general, the structure of the GIRAF system is based on the idea of separating and adjusting functionality. For the child using the system, only applications meeting the child's capabilities and handicaps should be available. And for the guardians responsible for administrating the system, all types of system and application settings should be available. Applying this separation helps the guardians set boundaries and adapt the system to the child.

Multi Project Management 3

In this chapter we describe how the multi project has been managed during the semester. First, in Section 3.1, it is described how different elements of the multi project, like code, documentation, and issue reports are shared between the four groups. In the same section, a brief introduction to Google Code is given, and different best-practices regarding use of branches in our shared code base are explained. Furthermore, Section 3.1 describes the different kinds of meetings that were held during the semester, including group leader and workgroup meetings for instance. In Section 3.2 the outcome of using the different multi project management tools is described, and finally, in Section 3.3, the outcome of the project related workshop in multi project management is covered.

3.1 Project Sharing

Since the development of this project was carried out by four independent groups, a structure for how to manage possible issues was needed. The following issues needed to be addressed:

- Source code sharing.
- Documentation sharing.
- Issue tracking.
- Inter-group communication.
- Project planning.

To address these issues, it was decided that two different tools should be used: Google Code (see Section 3.1.1) and meetings (see Section 3.1.2).

3.1.1 Google Code

In order to have a common way of accessing source code, documentation, and a list of registered issues, a Google Code Project named "sw6android: Android Project for Autistic Children" was created. A Google Code Project is an easy to use project management tool that can be used to organize the content of a project. Furthermore, it provides a web-interface enabling developers to browse the content of a project. It provides the following important services in regards to content management:

- SVN repository.
- Wiki.
- Issue tracker.

The SVN repository was used to manage the source code of the entire project. It was intended that each group's projects were stored in the SVN repository, meaning that at any time a developer should be able to

download the source code of another group's project, giving total transparency of the work. A group could have many different projects, and thus were allowed to create as many projects on Google Code as needed. Each group's folders in the SVN repository were mainly divided into a `branch` folder and a `trunk` folder. The `trunk` folder contained source code of the latest stable release from the group. The `branches` folder contained branches of a particular project - it was not required that branches contained stable code. Typically, a group created a `dev` branch, of a particular project, and used that as their main place for development on a particular project. Later, when the developed code was considered stable, the `dev` branch could be merged with the trunk, giving other groups the option to check out the new stable code, and use it.

The wiki page was used to share documentation, tutorials, and other project related texts. This gave all groups a centralized way of sharing project related content, making it easier to maintain an overview of the project while limiting misunderstandings. The issue tracker was used to publish known issues. When a group found an issue in another group's software, they would inform the other group by posting an issue report on the issue tracker.

3.1.2 Meetings

In order to plan and track the progress of the project, a series of meetings has been held during the project. There have been four kinds of meetings.

Group leader meetings At the start of the semester, each group chose a group leader who would represent the group at group leader meetings. The purpose of group leader meetings is to define the project scope, decide the features of the project, track the progress of the project, and handle issues related to the project in general.

Grand meetings A grand meeting is where every student from the multi project groups attend the same meeting to discuss the project in general. The purpose of such a meeting is to ensure that no persons feel left out in the decisions making.

Group to group meetings This kind of meeting have the purpose of discussing issues only relevant for the groups in question. They are typically short and have a limited topic scope.

Workgroup meetings Workgroup meetings are special comities setup to discuss specific technical issues in depth. Each group is represented at the meetings, not necessarily by their group leader, but by a student with most expertise in the technical field to be discussed.

3.2 Outcome

This section evaluates on the outcome of the tools used in the multi project management of this semester's project.

3.2.1 Google Code Outcome

Despite that one group did not honor the agreement of using the Google Code project, the Google Code project turned out to be a great tool for centralizing content from the groups. Source code sharing via the provided SVN repository turned out to be crucial, as both group sw6b and sw6d where developing libraries that the other groups needed to use. It was also beneficial that it was possible to track the progress of other groups, and make ad-hock compatibility testing by downloading the source code of other groups.

3.2.2 Outcome of Meetings

The group leader meetings went fine for the most part. Various decisions were made at the group leader meetings such as:

- Definition of target group.
- FACTORS for the multi project.
- System definition for the multi project.
- Source code license.
- Code conventions.
- The prefix: `sw6.*` should be used for every GIRAF package.
- Arrangement of workgroup meetings.

A single grand meeting was held. The meeting was organized by making an agenda containing topics that needed to be discussed. Every developer could propose topics to the agenda, to ensure that all developers got a chance to contribute to the project plan. However, the grand meeting did not go well. The main problem was that too many topics were on the agenda, and too many people wanted to express their opinions. Very few decisions were made at the meeting and most decisions were postponed to future leader meetings, due to lack of time, preparation, and pragmatism. However, the following decisions were made:

- We should use Google Code to centralize source code, project documentation, and issue tracking.
- We should follow the Google Android coding guidelines for Android development described in [And11d]. It was decided that minor modifications to the guidelines were allowed.

Other decisions were partially made, but because of disagreements and lack of preparation, the final decisions to these problems were postponed.

Group to group meetings were arranged ad-hock to deal with smaller issues that arose during the development. Because only a few topics were on the agenda at these meetings, they were often very productive.

Two workgroups were created during this multi project by request of the group leaders. One was created to discuss the overall architecture of the project. Two meetings were held discussing this issue, and a pragmatic solution was found (see the solution in Section 2.3). Another workgroup was created to make test designs and test cases to be used for integration testing of the modules in GIRAF. Furthermore, the workgroup was also responsible for executing the tests and report any issues found. Further details about multi project test is given Chapter 4.

During the project development, emails were used extensively. Whenever a group leader meeting had been held, a summary of the meeting was sent to all developers and supervisors, so that everyone could follow the decisions that were being made. Email was also used to arrange meetings, announcing important publications (such as new documentation documents and information about new stable releases), etc.

3.3 Multi Project Management Workshop

During this semester the MPL course was given with the purpose of teaching us tools that could be used for managing our multi project. Unfortunately, the course only gave us few tools to work with, as the focus of the course quickly turned away from multi project management onto internal development methods in the groups. The tools we have used from the MPL course are:

- FACTORS analysis.
- System definition.
- Component Architecture.
- Group leader meetings.
- Risk analysis.

The most useful tool turned out to be the group leader meetings. Having meetings with group representatives, instead of having grand meetings with all developers present, made it a lot easier to communicate between the groups and make decisions. As seen in Chapter 2, the FACTORS analysis and the system definition have been used to define the requirements for the multi project and as will be covered in Chapter 4, the component architecture has been used as a basis to form the design of the multi project test. Finally, as will be covered in Section 6.2.1.1, a risk analysis has been used to measure the risk in our part of the multi project.

Multi Project Test

4

As a part of the final work on this semester's multi project, it was planned to test if the multi project complied with its system definition seen in Section 2.2. Therefore, the multi project leaders arranged that a workgroup consisting of one person from each group should schedule a couple of workshops where test planning and test execution for the multi project should be carried out. In this chapter, we describe the test approach, the test execution, and the outcome of the multi project test.

4.1 Approach

At the first meeting in the workgroup, an overview of what should be tested was made. It was decided to base the tests on the system architecture of GIRAF seen in Section 2.3, and use that as the starting point for defining an overview of what should be tested. The outcome of this process is the overview seen in Figure 4.1.

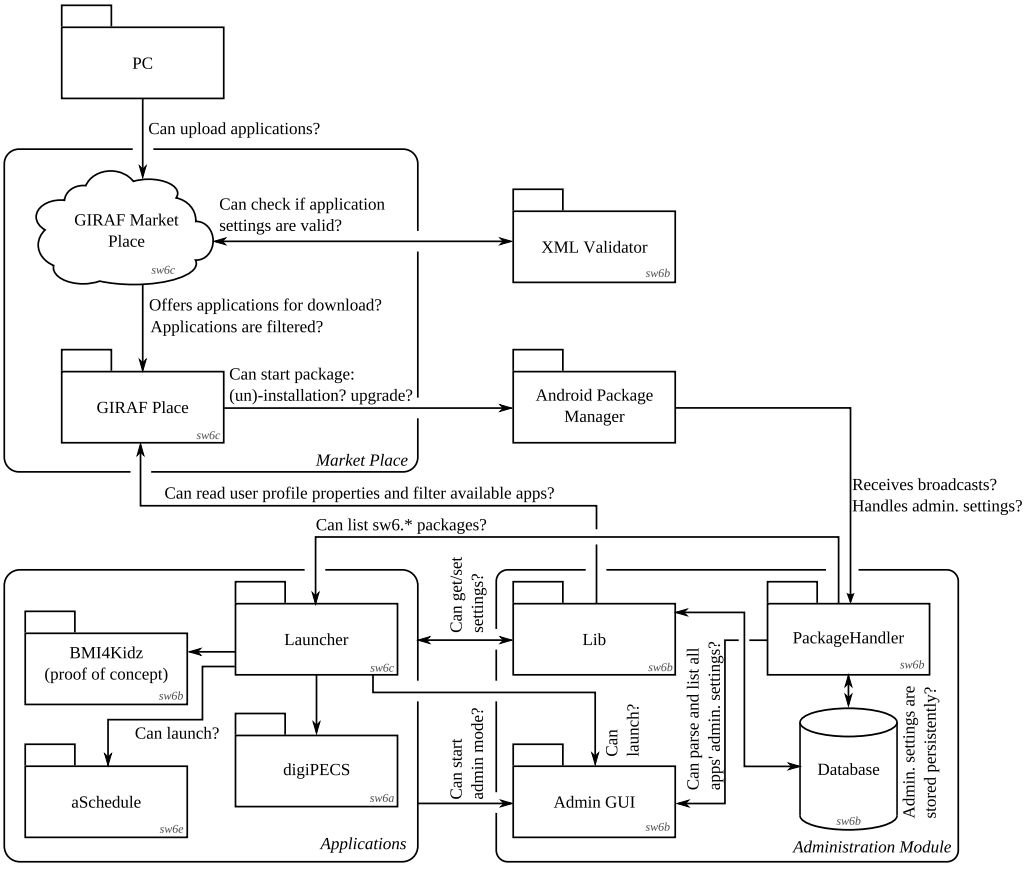


Figure 4.1. Shows the main structure of what should be tested at the multi project integration test.

On the basis of Figure 4.1, a list of concrete test designs and test cases were made. To ensure that the whole system was covered by the written tests, Figure 4.1 and the system definition was continuously taken into account. In Table 4.1 the test objectives of each of the planned test designs are seen. In Appendix D.6 a more detailed list of the defined test designs and test cases can be seen.

Identifier	Test Objective
TD00	Ensuring GIRAF applications can be deployed to the GIRAF Market Place, and non-GIRAF applications as well as corrupted GIRAF applications are rejected.
TD01	Testing initial installation of GIRAF, and that the user is presented with necessary welcome dialogs.
TD02	Ensuring applications can be installed on different Android devices using the GIRAF Market Place client. Ensuring that the launcher and GIRAF applications are able to get and set settings in the administration module of GIRAF. Ensuring that the launcher and the GIRAF applications abide to a functional navigation flow when opening "in-app" administration mode.
TD03	Installation and upgrade of GIRAF applications.
TD04	To test the back button behavior of GIRAF Launcher and applications.
TD05	To test whether the home key handling works as intended.
TD06	To test if applications can be deleted using the GIRAF Market Place client.
TD07	To test if applications are filtered correctly in the GIRAF Launcher and the GIRAF Market Place client according to user profile settings.

Table 4.1. List of objectives for each of the planned test designs.

At the first meeting in the multi project test workgroup, it was planned that the workgroup should be responsible for carrying out the tests, and that one workday for test case execution should be reserved. Based on this, the type of test that was planned to be executed was a dynamic black box integration test. Basically, this means that integration of the multi project modules should be tested by some testers using the system while checking if the system complies with some specified requirements. In this process, the testers should only use the system from the outside, and not look at the implemented code. In Chapter 12, the tests carried out in our module of the multi project (the administration module) are described. Here, terms like static- and dynamic black- and white box testing, and a series of other topics in the field of software testing will be covered and described in detail.

4.2 Execution

In this section we describe the test-case execution as well as different problems that occurred during the preparation.

Three days before the test execution, an email was sent to every group in the multi project with instructions on what to prepare for the multi project test day. For instance, it was required that each group had made a working revision of their project available through our common Google Code SVN repository. However, the day where the tests were to be carried out, the group responsible for the PECS application (sw6a) had not committed working code to the repository. As will be covered in the conclusion on the multi project (see Chapter 15), the PECS group refused from the early phase of the multi project to use the Google Code project site and to meet the common use of Integrated Development Environments (IDEs) that was agreed upon at the very first meeting of the multi project. Even though that the other groups in the multi project expressed their concerns about this, the PECS group continued refusing which led to problems during the multi project test. After realizing that they had refused to use Google Code even for the purpose of the multi project test, a couple of hours went discussing why it was important for the multi project that they shared their project at Google Code. Finally, a revision was released of their application. However, as a consequence of being

very uncooperative and refusing to meet the coding guidelines defined and agreed upon at project start, no one was able to compile the code of the PECS application, as different configurations files were missing. After having struggled with the code for some time, the PECS group build their project on their own, and provided a binary of their project. However, it was soon clear, that the binary did not include the latest revision of the administration and launcher library, rendering the binary of the PECS application useless. After hours of struggling with the PECS application, the conclusion was to disregard it, meaning that the PECS application *has not* been a part of the integration test for the multi project.

At the test, the following applications were ready to be tested:

- aSchedule
  - Is an easily managed electronic visual schedule for children with autism and their guardians, developed by group sw6e.
- GIRAF Place
  - Is the market place for GIRAF applications, developed by group sw6c.
- GIRAF Launcher
  - Is the home screen of the GIRAF system when running on an Android device. It enables filtering of applications so that only applications meeting the abilities of the user are shown. The launcher is developed by group sw6c.
- Administration Module
  - Handles administrative settings for GIRAF applications. Is develop by group sw6b. The administration module is described in detail in Part II.
- BMI4Kidz
  - Is a demo application developed with the purpose of testing most of the functionality available in the administration module of GIRAF. The application is developed by group sw6b, and is described further in Chapter 8.

The tests covered in Table 4.1 were executed one by one, and the observations and results were written down together with the test cases, see Appendix D.6 for details. Also, each issue was reported using the issue tracker at the Google Code project site.

4.3 Results

In this section the results of the multi project test are described. In total, eight test cases were executed, with a count of 57 different steps. Nine issues were found and reported. In the following, the issues found are described and the current status (May 24th 2011) of the issues are covered too.

TD00, Google Code Issue No. 44 The GIRAF Market Place does not check if the administrative settings of an application are valid. This should be checked on upload of an application. *Related to: group sw6c. Status: fixed.*

TD01, Google Code Issue No. 45 Rotating the phone when editing an administrative setting, like the name of the child, clears the content of the edit text box. *Related to: group sw6b. Status: fixed.*

TD01, Google Code Issue No. 46 The cursor in an edit text box cannot be manually placed to the right most position, when editing a text field like "Name" in the GUI provided by the administration module engine. *Related to: group sw6b. Status: fixed.*

TD01, Google Code Issue No. 47 The administration module loads slowly on an HTC Wildfire. *Related to: group sw6b. Status: will not be fixed, as the Wildfire is build for Android 2.1 and only runs 2.2 due to installation of a custom Read-only memory (ROM). Cannot be reproduced on the Android 2.2 hardware provided by the department.*

TD02, Google Code Issue No. 48 Icon of the "Refresh" functionality in the GIRAF Market Place client is too big. *Related to: group sw6c. Status: will not be fixed. The group does not know how to resolve this issue.*

TD03, Google Code Issue No. 49 List of applications are not refreshed automatically when resuming the use of the GIRAF Market Place client. *Related to: group sw6c. Status: fixed.*

TD03, Google Code Issue No. 50 Google Android icon guidelines not followed. *Related to: BMI4Kidz and the administration module, both developed by group sw6b. Status: fixed.*

TD03, Google Code Issue No. 51 An upgraded application is shown twice in the home screen of the launcher. *Related to: group sw6c. Status: fixed.*

TD05, Google Code Issue No. 43 Application state saved when entering the administration module from within an application. This allows access to administration mode without entering the "secret key combination". *Related to: group sw6b. Status: started. Different solutions have been discussed, and a final will be applied before project hand-in.*

The number of issues reported indicates that the multi project test has paid off, as very crucial functionality was observed to not function correctly. Besides the issues found and the fact that the PECS application could not be tested, the test case execution showed that the life cycle of installing applications with valid settings, using the applications, edit their administrative settings, and so on, should have a great chance of working in the final release of the multi project this semester. This shows that the groups of the multi project (some more than other) have tried to make an effort to ensure that the integration of modules should be one of the strengths in GIRAF. Based on this, it seems reasonable to assume, that the GIRAF multi project, before hand-in, will be able to meet most of its requirements specified in its system definition defined in Section 2.2, and that the system architecture seen in Section 2.3 will be reflected.

5 Technical Details of the Software Target Platform

The purpose of this chapter is to provide a technical description of Android, as it has been chosen as the software platform of the GIRAF system - see Section 1.3.1 and Section 2.2. The chapter is not intended to give a complete description of the entire platform. Instead the aim is to provide a description of the technical parts of Android that were used during development of the administrative module and the associated library. Because Android is available in several different versions the chapter will be limited to Android 2.2.

5.1 Overview of the Android Platform

Android is not an OS in itself, but more a collection of many different programs and modules. Android uses a slightly modified version of the Linux kernel (Linux 2.6.32 in Android 2.2) that takes the responsibility of providing an abstraction layer between the hardware and software on the Android device. On top of the Linux kernel, multiple layers reside with different components. This is illustrated in Figure 5.1 showing the system architecture of Android.

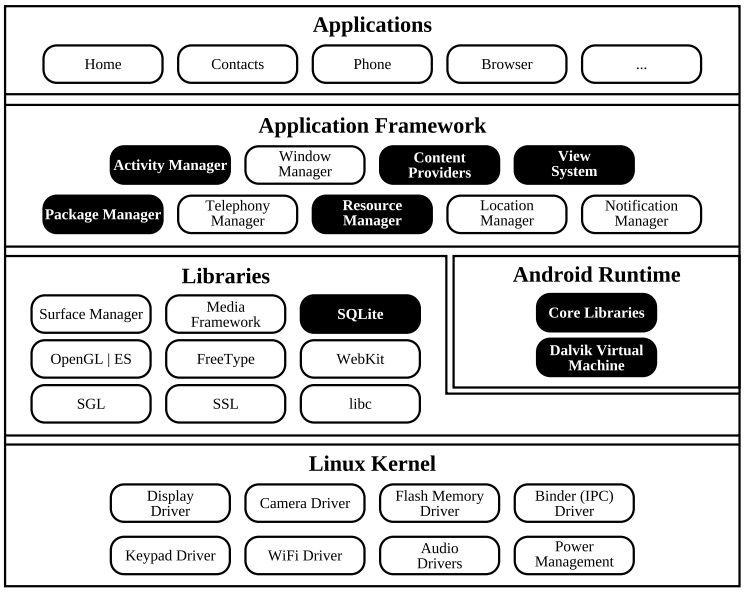


Figure 5.1. Major components of the Android operation system. The highlighted components (the black boxes) show the components we have used in our part of the multi project.

The following sections will deal with the different layers in the Android architecture and describe the components of each layer that is used during the development of our project. The source used is [And11l] unless otherwise indicated.

5.1.1 Android Runtime

This layer contains the core functions that Android applications need in order to execute, and call basic functions on an Android device. The purpose of the runtime layer is to provide an abstraction so that developers do not have to target the Linux kernel directly. The Android runtime layer is divided into two primary components where each is important for the system:

5.1.1.1 Core Libraries

The purpose of the core libraries is to provide a common functionality that all applications have access to. The functionality provided by the core libraries is a subset of the functionality provided by Java. In other words, a developer does not have access to the entire Java library, but only the subset that Android provides. Therefore, it is not all Java modules that can be moved directly into Android applications without modification.

5.1.1.2 Dalvik Virtual Machine

Even though Android applications are written in the Java language it is not Oracle's Java Virtual Machine (JVM) that is used to execute the applications. Instead the Dalvik Virtual Machine (DVM) is used. DVM is designed specifically to use few resources and run many instances of virtual machines simultaneously with little overhead. This is useful on the Android platform because it allows each application to run in its own DVM. Furthermore, DVM outsources process handling, memory handling, and thread management to the Linux Kernel. A major difference between DVM and the JVM is that DVM does not execute Java bytecode but instead executes its own binary format `.dex` (Dalvik Executable). In order to create `.dex` files, a tool called "dx" is used to convert Java bytecode to one or more `.dex` files.

5.1.2 Libraries

Android includes a set of various libraries written in C/C++. The libraries are used by different components of the Android OS. One of these libraries is the SQLite database engine. In the following, we briefly describe the functionality of this library.

5.1.2.1 SQLite

Android offers multiple ways for applications to store data. For example, it provides the possibility of storing data in SQLite databases. Unlike many other kinds of SQL databases, SQLite does not use a server process but instead operates directly on flat files. Therefore, each database is a single file in the file system. On the Android platform, database files are placed in the application's own private data directory, which can only be accessed by the application itself. If an application needs to share its database, this is done using a content provider - described further in Section 5.1.3.4.

SQLite allows the developer to perform almost all standard database operations. It allows the creation of tables, insertion/deletion of rows, updating of rows, and finally it also allows the developer to make queries for retrieving rows. Android 2.2 comes with SQLite 3.6.22, unlike previous versions which came with SQLite 3.5.9 [Car11]. The main difference between the new version and the old version is the ability to create constraints across tables, such as foreign key constraints [SQL11c].

5.1.3 Application Framework

It is possible to create applications that only use core libraries, but they will not be able to be started from a launcher. Therefore, the application framework contains all the components needed to create real Android applications that can be used by Android users and interact with other applications.

5.1.3.1 Intents

Intents are not included in Figure 5.1 showing the system architecture of Android. Even though, they are quite important to Android as they act as a link between the various components in applications. The purpose of intents is to send messages between components and ensure that the relevant activity starts up or is informed of various events. An activity can specify what types of intents they accept, and Android will keep track of this. Therefore, when an application sends an intent, Android will know which activities can handle this. In the event that multiple activities can handle the same intent, Android will prompt the user asking which activity should handle the intent. An example of this is seen when there are several applications that can handle URL's. This will result in the user getting a list of installed browsers to choose from. However, in this situation it is possible to select a default setting such that the user is not prompted with the same question in the future.

Broadcasts
A special type of intents is called Broadcasts and is created if a component needs to tell the rest of the Android system that something has happened, but expects no response back. Android will then find all broadcast receivers that listen to this signal and start them up to deliver the signal. An example of one of these broadcasts is when an application is installed on the Android device. This will result in a `android.intent.action.PACKAGE_ADDED` broadcast being sent by the Android Package Manager which then can be received by a broadcast receiver.

5.1.3.2 Activity Manager

Most Android applications consist of at least one activity. An activity can be described as a screen on the phone and when switching between different screens different activities are swapped in and out. However, it is possible to have activities that do not contain any GUI elements and which are just invoked as a kind of method. Activities can also be marked with various flags (e.g. the "main activity" flag or the "display in launcher" flag). If the "main activity" flag is set for an activity, the activity will be the main starting point of the application. If the "display in launcher" flag is used, the flagged activity will be visible in the launcher[1] of the Android device. So typically, to allow an application to run and be launched, many applications will include these two flags.

The purpose of the Activity Manager is to keep track of the different activities that applications offers and which intents they accepts. Controlling this enables Android to look up in the Activity Manager instead of going through all installed applications, when intents are sent through the system. At the same time, the Activity Manager keeps track of the lifecycle of the various activities such that Android can determine which activities can be cleared out of memory.

[1] A launcher is that part of Android that is used to start applications from. Normally, a launcher is attached to the home-button of the Android device, such that the user easily can start launching applications.

5.1.3.3 View System

This component contains a number of standard views that can be used by applications to provide various kinds of standard GUI setups. Theses standard GUIs include *Linear Layout*, *Relative Layout*, *Grid View*, *Layout Tab*, and *List View*. An example of using a *List View* can be seen in Figure 5.2.

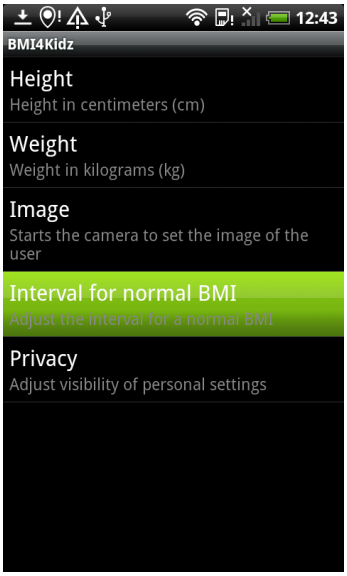


Figure 5.2. An example of how a *List View* can be used to list different settings for an application.

It is also possible to nest and combine different views, such that a view is divided into several parts where each part contains a different view.

5.1.3.4 Content Providers

Section 5.1.2.1 described that applications cannot access private data of other applications. In order to share data between applications, content providers are needed. An application can offer a content provider which serves as a data communication link to other applications. Communication can be done by sending the following inquiries:

Query Asks for data that meets certain criteria specified by parameters.
Insert Inserts information specified by parameters.
Update Updates specific data, determined by a set of criterion.
Delete Deletes the data that meets a specified criterion.

There is great similarity between a content provider and the interface a typical SQL database provides. Therefore it is also possible to communicate with a SQLite database directly through a content provider. Even though, this is still something the developer has to offer if other applications should be allowed to access the database of the developer's application. The principle of content providers can be seen in Figure 5.3.

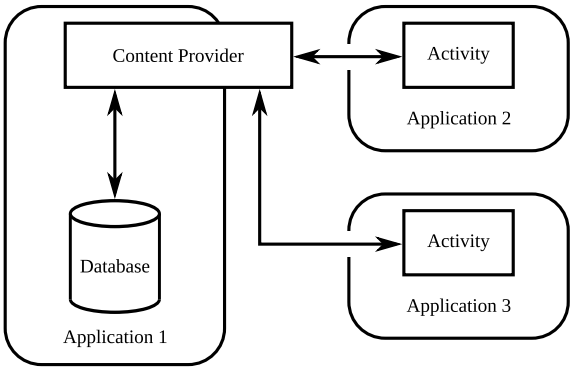


Figure 5.3. A content provider provides an interface that can be used by other applications for accessing data in the application wherein the content provider is declared.

A content provider can be used by any application on the system that has knowledge of it existence. The benefit of content providers is the ability to specify exactly what actions are allowed and which are not allowed. If, for instance the developer decides that it should not be possible to erase data, he can simply neglect to implement this. It is also possible to perform input validation so no invalid data is sent to the database. In the Android platform there are already defined several content providers providing a simple interface between applications and public available data. As an example, the Android platform defines a content provider for the phone book and makes it possible for an application to resolve a number to a specific contact and get the contact's name [And11e].

5.1.3.5 Package Manager

All installed applications on the Android device are administered through the Android Package Manager. The Package Manager is invoked when an application needs to be installed, uninstalled, upgraded, or have its data cleared. An application can also ask the Package Manager for information about various packages. The Android Package Manager sends out broadcasts, whenever something within the package manager has happened. Examples of the broadcasts are:

`android.intent.action.PACKAGE_ADDED`, is sent after a package has been installed.
`android.intent.action.PACKAGE_REPLACED`, is sent after a package has been upgraded.
`android.intent.action.PACKAGE_REMOVED`, is sent after a package has been removed.
`android.intent.action.PACKAGE_DATA_CLEARED`, is sent after all data for an application has been deleted.

5.1.3.6 Resource Manager

Android applications do not only include code but often also different resources such as graphics, translated strings, and layout files are included. The Resource Manager of Android provides access to these resources. Furthermore, Android contains a long series of standard resources that can be accessed, e.g. like standard graphics. By using the built-in resources, applications will automatically integrate with the rest of the theme on the phone.

5.1.4 Applications

The top layer of the Android system architecture seen in Figure 5.1, contains the individual applications that are installed on the Android device. An Android device comes with a variety of installed applications, e.g. a call application. It is possible to start applications from other applications as long as all involved applications are installed on the device. It is in this top layer of the Android system architecture, that GIRAF will reside.

5.2 Android Development

Android is an open platform that comes with a free SDK, available for Windows, Linux, and Mac. Besides providing an Application Programming Interface (API), a wide range of useful tools are provided helping with development of applications for the Android platform. This section will address some of the tools used during development of our project. In the end there will be a description of Android Package (APK) files and how Java source code is translated into an APK file.

5.2.1 Android SDK and AVD Manager

The Android SDK consists of many components so it can be difficult to manage them all. Therefore, Google Inc. has developed the program "Android SDK and AVD Manager" to manage all components of the Android SDK. The program is installed together with the Android SDK, which then allows the developer to download different versions of the Android SDK depending on which version the developer wants to develop for. Furthermore, when the program is started, it checks for new versions of the installed components [And11a]. A usage example of the program can be seen in Figure 5.4.

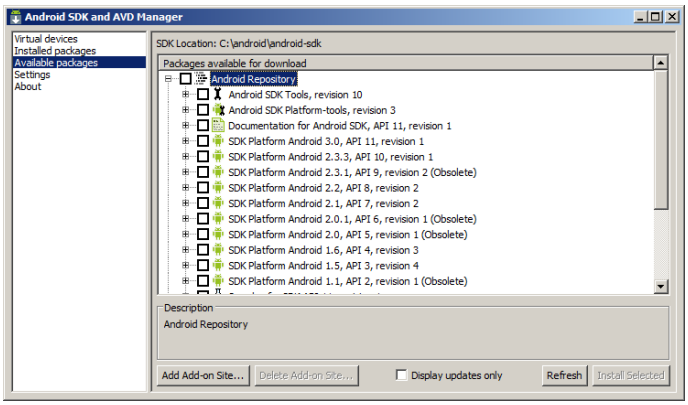


Figure 5.4. The Android SDK is administered through the Android SDK and AVD manager. The figure shows a portion of the list of possible components that can be installed.

5.2.2 Android Virtual Devices

The "Android SDK and AVD Manager" program is also used to administer Android Virtual Devices (AVDs), which serves as virtual Android devices which developers can use when they do not have access to a physical device. An AVD simulates a pure Android device without extra additions from other manufacturers. When creating a new AVD, it is possible to specify parameters such as Android version, screen resolution, and amount of ram [And11g]. An AVD is very useful when developing applications, but it is still necessary to test applications with real physical devices as there may be product-specific differences between the Android versions used. Moreover, an AVD is controlled by a mouse and a keyboard which is not suitable for simulating multi-touch. From a performance point of view, AVDs are problematic as they execute applications a lot slower than newer physical devices. Figure 5.5 shows an example of an AVD running Android 2.2.

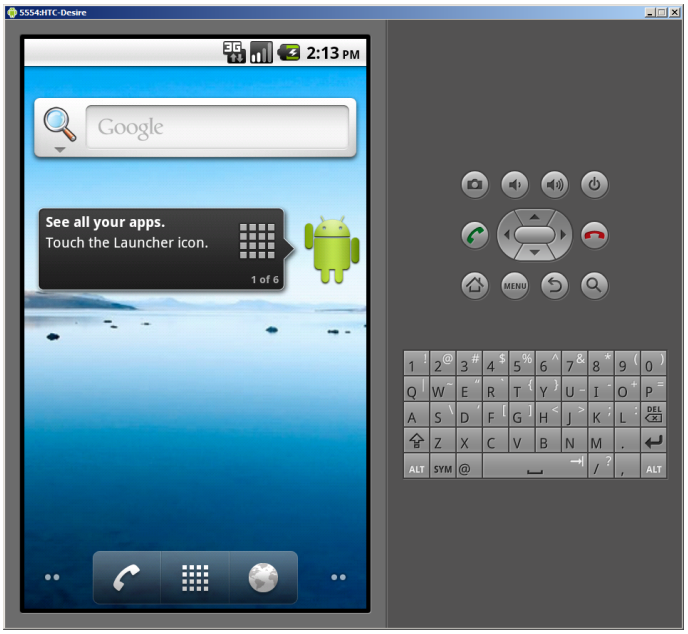


Figure 5.5. An AVD simulating an Android 2.2 device. On the right panel the developer can access all the buttons that typically are available on an Android device. Using external tools, it is also possible to simulate various operations/events such as receiving an SMS.

One advantage of AVDs is that they are less restrictive than most manufactured phones. Most phones do not have access to files and folders in Android making it impossible to browse phone content from either the phone or from a connected computer. An example of this restriction is, that it is not possible to access private data of various applications. The database for an application is placed in a private data folder unique for the application in question, and the folder cannot be directly accessed on the phone or from a computer. During the development of applications it is often necessary to access the database directly, e.g. for debugging purposes. However, as the database is stored in the application's private folder, it is per default not possible to gain external access to the database file if it is stored on a physical device. In order to access these private files, the physical Android device has to be "rooted"[2]. However, the device manufacturers rarely allow this, so as an alternative, AVDs are used since they provide read and write access to private data.

5.2.3 Android Developer Tool

Android Developer Tool (ADT) is an extension to the Eclipse IDE which integrates various Android tools directly into Eclipse. ADT extends the existing functionality in Eclipse [And11b], allowing:

- Rapid creation of new Android projects.
- What You See Is What You Get (WYSIWYG) creation of GUIs in Android applications.
- Debug of applications using the Android SDK tools.
- Export of Android projects to APK files.

Early in the development process, it was decided that all groups should use "MOTODEV Studio for Android" which is an IDE based on Eclipse that comes with the ADT plug-in. Moreover, the product contains other useful tools such as code snippets which are small and simple code examples.

[2]Rooting a device is a process where the security is modified such that the user is able to execute programs with root privileges. The root user has permission to do anything on the Android OS, and is therefore also able to access all resources and modify protected system data [Fre11].

5.2.4 Android Package

An Android application is installed on an Android device using an APK file. An APK file is basically an archive containing different files and folders. Usually an APK file contains a single application but it is possible to have multiple applications in one APK package. Administration of APK files is handled by the Android Package Manager and can be accessed both via a GUI on the Android device and through interfaces in the Android API. This makes it possible for users and developers to access information about installed packages. APK files are created by completing a series of processes which are briefly illustrated in Figure 5.6 [And11c].

Figure 5.6. Shows the process of how an Android project is turned into an APK file that can be installed on an Android device.

An Android project consists of source code and associated resources. During the "Compilation and Packaging" process, all source code and resources are packed together into a single APK file. An APK file contains the following elements:

.dex files During compilation, all source code in the form of `.java` files are compiled into `.class` files. After that, the `.class` files will be compiled into Dalvik byte code and combined in one or more `.dex` files. Dalvik byte code is executed in an instance of the DVM running on the Android device.

Resources Resources belonging to the application are placed in the APK file. Examples of resources include images, strings, and layout files in XML format. During construction of the APK file, Android will try to optimize all resources by limiting their size and making XML files fast to load by converting them to binary code.

AndroidManifest.xml All Android applications contain this file which is located in the root of the APK file. The file is written in XML and contains all information about the application that Android needs. The following is an overview of some of the information an `AndroidManifest.xml` file can contain. All information comes from [And11k] unless otherwise stated:

- The package name for the application. This name must be unique for each application on the phone since this name is used to identify the application.
- Description of various components the APK file contains. This applies to activities, services, broadcast receivers, and content providers. The descriptions are for example used by Android to create a list of which applications that should receive broadcasts when an application has been installed or upgraded on the system. Because of this list, Android does not have to go through every application checking if they can receive broadcasts, every time a broadcast is sent.
- Permissions needed to install the application. Many features in Android require the user's accept. For example, an application can specify in the `AndroidManifest.xml` file that it must have write permissions to the SD card. During installation of the application the user is informed of these requirements, allowing cancellation of the installation.
- The minimum version level of the Android API the application requires. Each version of Android has its own API level (e.g. Android 2.2 has level 8) [And11i].

After the APK archive has been built it is signed with a key. There are two types of keys: *debug* and *release*. The debug key is automatically generated by the IDE and used to sign applications that are being continuously developed and uploaded to the device. The debug key gives permission to install the application via Universal Serial Bus (USB) or via manual installation, however the debug key cannot be used to sign applications for Android Market. To do so, a release key must be used. A release key is generated for each developer. A developer can then sign an application with the release key enabling the user to see who signed the application. Common to both keys are that if an application is to be upgraded, then the application can only be upgraded if the new package has been signed with the same key that was used to sign the old version of the package. When the APK file has been signed, it is ready to be installed on an Android device.

Part II

Administration Module

This part describes the administration module developed by our group, and the development method used during the development process. Chapter 6 describes the development method. A detailed technical overview of the administration module is presented in Chapter 7. Chapter 8 describes how an application is integrated with the administration module.

The administration module provides a library that applications can use to read and modify application settings. This library is described in detail in Chapter 9. In order to provide this service for applications, a wide range of technologies have been used and developed. An SQLite database has been created to store the actual values of settings persistently. In order to communicate with the database, a database helper has been implemented, serving as an abstraction for Structured Query Language (SQL) commands. This, and other technologies like content providers, broadcast receivers, and package handling used in the engine of the administration module are described in Chapter 10.

In Chapter 11, the GUI of the administration module is described. The GUI enables a guardian to browse and modify administrative settings of GIRAF applications. Finally, Chapter 12 describes various types of tests and code coverage analysis that were performed in our project. Test approaches like unit-, regression-, and automated integration testing are described, along with an introduction to how code coverage analysis can be performed on the Android platform.

Development Method

6

In this chapter we describe the development method used during development of the administration module. First, Section 6.1 describes the development method. It should be noted that the section should be seen as a reference work. After that, in Section 6.2, we give a rational for why the development method has been chosen. Here, topics like agile development and risk analysis are covered and taken into account. It should be noted that an evaluation of how the use of our development method went is presented in Chapter 13, located in the recapitulation part.

6.1 Description

We have not chosen a common development method like Extreme Programming (XP) or Scrum. Instead we have decided to create our own method. Our development method borrows elements from existing development methods and it consists of different techniques and structures that will be described in the following sections.

6.1.1 Daily Stand-up Meeting

Daily stand-up meetings will be held at 09:00AM, sharp. A moderator is elected in the initial phase of the project, who will mediate at the meetings. At the stand-up meetings, the developers must be asked the following four simple questions:

1. *What have you done since the last meeting?*
2. *What will you do between now and the next meeting?*
3. *What is getting in the way (blocks) of meeting the iteration goals?*
4. *Have you learned or decided anything new of relevance to some of the team members? (technical, theoretical, etc.)*

It is important that all developers physically stand up at the meeting. This will force the participants to make the meeting short, as it is tiresome to stand up for longer periods.

6.1.2 Development Process

Development will be based on a bottom-up approach, with no big design up front. Programming work is structured using iterations, where a certain number of tasks are planned to be finished within the time interval of an iteration. An iteration starts on Friday afternoon, and ends Friday morning. At Friday afternoon, the iteration for the next week is planned. The iteration will formally be written down in a wiki article and shared using the wiki system at the Google Code multi project site. At Friday morning, the goal is to ensure that

everything is compiling, all tests succeed, and that the code is committed to the trunk repository as a release. If for some reason we are having trouble meeting the deadline of an iteration, normal development practice dictates that the scope of the iteration should be limited. However, in our development method, no one goes home on Friday before the entire iteration has been finished. After each iteration, and before the next iteration starts, each developer must present what has been developed. The presentation is given to the other developers in the group and helps everybody understand what is happening and how the implementation works.

While developing the project, a developer may at any time ask another developer from the group to review some code. It is not mandatory to make a code review every day, however before the project is handed in; all code must have been reviewed by all members in the group. The group itself will be responsible for this. Furthermore, ensuring this also helps every developer to fully understand what is written, and how the system works in detail.

6.1.3 Testing and Documentation

Testing will be done continuously throughout the project using unit tests. Our primary focus will be on testing the part of our software that other groups will use. Furthermore, integration testing will be done continuously. See Chapter 12 for details regarding how testing has been carried out in our project.

Code documentation will be written as Javadoc/Doxygen compatible comments in the source code. A central documentation document, documenting the source code of the project, will then be generated using the Doxygen tool, which is further described in Section 9.4. However, basically the tool can be used to generate HyperText Markup Language (HTML) documents displaying the code documentation in a structured way. All other documentation and "getting-started" tutorials will be written in text and published in PDF to our fellow students. It should be noted that, as Google Code do not allow us to show documentation in HTML format and group content related to a specific module, we created a project site for the administration module. The project site is found at: `http://sw6b.lcdev.dk`, and we used it as our primary distribution channel for new documentation documents. To ensure that every student in the multi project became aware of new material and changes made in existing documentation, we sent emails to every developer in the multi project when our project site had been updated.

6.2 Rationale

This section will give a rationale for why we have chosen the development method described in Section 6.1. Our arguments will be based on the circumstances of this project (e.g. the size of our team) and a risk analysis of the project. First, "the one-minute risk assessment tool" is presented, and used to carry out a risk analysis for our project. Based on the results from the risk analysis, we give the rationale for our development method.

6.2.1 Risk Analysis

As a part of choosing the methodology for our development method, the risk involved with the project was first estimated. This was done using "the one-minute risk assessment tool" described in [TK04]. In this context "risk" means the risk that something goes wrong in the project (e.g. project delay due to bad management or software that does not satisfy what the customer had in mind). This tool was given to us in the Software Engineering (SOE) course, that we have attended this semester. The reason why we used this tool was to identify possible risk problems that needed to be addressed in the creation of our development

method. The risk analysis performed in this section will serve as the main rational for our development method.

"The one-minute risk assessment tool" is created by Amrit Tiwana, (assistant professor in the Goizueta Business School at Emory University, Atlanta) and Mark Keil (Board of Advisors Professor of computer information systems, J. Mack Robinson College of Business, Georgia State University, Atlanta) by analyzing 720 software projects from 60 large companies focusing on the cause when something went wrong in a project. They concluded that the most likely cause that something went wrong was the use of an inappropriate development methodology. The next most likely cause was lack of customer involvement, followed by lack of formal project management practices, dissimilarity to previous projects, project complexity, and at last; requirements volatility. Based on this study, a formula for calculating the risk involved in a project was created by the two professors. The formula can be seen in Table 6.2, with reading guidelines listed below.

1. On a scale of 1 - 10, where 1 is low and 10 is high, how would you characterize this project compared to other projects completed in your organization?
2. Add the six weighted ratings (see the worked example in Table 6.2).
3. A lower overall project risk score indicates higher project risk. Range: 10 (most risky) to 100 (least risky).
4. Use Table 6.1 below as a guide for interpreting of this score.

Overall risk score →	10-28	29-46	47-64	65-82	83-100
Project risk level →	High	Moderately High	Medium	Moderately Low	Low

Table 6.1. Interpretations of risk scores.

Project Characteristic Question	**Rating**	×	**Weight**	=	**Result**
Fit between the chosen methodology and type of project		×	3.0	=	
Level of customer involvement		×	1.9	=	
Use of formal project management practices		×	1.7	=	
Similarity to previous projects		×	1.5	=	
Project simplicity (lack of complexity)		×	1.1	=	
Stability of project requirements		×	0.8	=	
Overall project risk score (higher score indicates lower project risk) →					____

Table 6.2. Template for calculation of project risk score.

The formula is used by grading the six different areas of the project from 1 to 10. This grade is given relative to previous projects. After that, a weight is multiplied with the grade. Every area has its own weight, calculated by using statically data from the analysis of the 720 software projects. The areas with the highest weight have the highest impact on the risk. When the weights have been multiplied with the grades, the results are summed resulting in an overall project risk score. The lower the score, the higher the risk. A scale is given in Table 6.1 that can be used to assess if the risk is high, moderately high, medium, moderately low, or low.

The purpose of using this tool is to get a rough estimate of how high the risk is with a given project. If the risk is high, the project may be too dangerous to start. If the project risk is moderately high or medium, certain measures should be taken to counter the areas that are causing the risk. This is done by looking at the six grades given. If for example the "Stability of project requirements" area has a low grade, measures must to be taken to ensure that this does not result in project problems.

6.2.1.1 Risk Analysis of Our Project

The risk in the project that our group is going to develop is estimated using this tool. Our estimations can be seen in Table 6.3.

Project Characteristic Question	**(Rating**	×	**Weight**	=	**Result**
Fit between the chosen methodology and type of project	8	×	3.0	=	24
Level of customer involvement	6	×	1.9	=	11.4
Use of formal project management practices	8	×	1.7	=	13.6
Similarity to previous projects	3	×	1.5	=	4.5
Project simplicity (lack of complexity)	3	×	1.1	=	3.3
Stability of project requirements	3	×	0.8	=	2.4
Overall project risk score (higher score indicates lower project risk) →					59.2

Table 6.3. Calculation of project risk score for our project.

We give the area *"Fit between the chosen methodology and type of project"* the grade 8. Compared to previous projects our development method fits the project very well.

We give the area *"Level of customer involvement"* the grade 6. As our group is developing the administration module for GIRAF, the customer, seen from our perspective, can be seen as the other groups. The grade is not high, as they are not "on-site" customers that always are available in our group room. However, they can always be contacted, and this is why we rate this with a medium grade.

We give the area *"Use of formal project management practices"* the grade 8. Compared with previous projects we are using a lot more formal project management practices. We are for example using iterations to schedule tasks, and daily stand-up meetings to ensure all developers are informed of what has been done, and what will be worked on.

We give the area *"Similarity to previous projects"* the grade 3. This project is very different from previous projects. We are developing for a software platform that none of us has any experience with. Furthermore, we have never worked in a multi project before which posses' great risk. The only reason this area does not get the grade 1, is that we are developing the software using the Java programming language, which all group members have previous experience with.

We give the area *"Project simplicity (lack of complexity)"* the grade 3. The project is complex compared to previous projects. The main reason for this is that we are developing a software module for a larger software system. This means that we have to cooperate with other independent groups, which results in a lot of unknown factors (like, are they willing to cooperate at all?). Furthermore, while developing, we will have to make sure that our software module will be compatible with the other software modules of the multi project. This makes the project complex since the other software modules are being developed in parallel with ours.

We give the area *"Stability of project requirements"* the grade 3. Since the requirements for the administration module are defined ongoing, the stability of requirements is low. Requirements may be added or changed at any time. The grade is medium-low, as we assume that when requirements have been specified, they will not be subject to major changes.

The overall risk score of our project has been calculated to the value 59.2 which qualifies the project as a medium risk project. As stated earlier, if the project risk is moderately high or medium, certain measures should be taken to counter the areas that are causing the risk. In the following, we take those areas into account.

The first area is *"Similarity to previous projects"*. A lot can be done from our side to remedy this problem. Using a bottom-up approach in our software development will help reduce the impact of this risk. Instead of

making big design documents at the beginning for the software system, we instead design the software from the bottom and up as we develop. Doing this allows us to develop the software while becoming confident with the new platform.

The second area is *"Project simplicity (lack of complexity)"*. This is mainly because this is a multi project where we are developing a software module for a larger system. This problem will be addressed by defining a structure for how the multi project should be managed. This structure is described in Chapter 3.

The third area is *"Stability of project requirements"* which is only a little problematic. It has a low grade but also a low weight making it less important. But it is still a risk factor that needs to be handled. The risk is again remedied by using an agile approach to development. Making a big design for our software in the start will cause us problems when project requirements are changed. This may result in us having to change large parts of the design, if not all. Instead we develop the software using a bottom-up approach, creating the design ongoing.

Using "the one-minute risk assessment tool", we can conclude that it is crucial that we use an agile development method in order to handle the risks involved with the project. However, because of the small size of our development team, the classical agile methods XP and Scrum are not suitable. Because of this, we have decided to create our own development method, borrowing ideas and approaches from existing agile methods (mostly XP). Using this result, we give a rationale for the elements of our development method in the following sections.

6.2.2 Daily Stand-up Meetings

Daily stand-up meetings with pre-defined questions for the developers, gives the opportunity to get an overview of how the work has progressed the past day. This is good for all members in the project group. A developer gain expertise in understanding and explaining his own code, and might find bugs when he is trying to look at the code from another perspective. For the audience developers, such a presentation helps them understand what is going on, and helps them gain a better view of how the system is currently working. Furthermore, the daily stand-up meetings give the opportunity to plan what is to be done the current day, and gives an overview of who is working on different tasks.

6.2.3 Development Process

Based on the result from the risk analysis stating that an agile development approach is suitable for our project, we have decided to make use of iteration-based development, which is a concept we have borrowed from the agile development method XP. We have chosen an iteration length of one week, in order to be able to adapt our development to project requirements that might change often during the multi project. The reason why an iteration should start and end on a Friday is, that typically you always work hard the last day of an iteration to meet its scope. And by placing the last day on a Friday, we do not risk that night work ruins an everyday where we have to attend courses. The reason why we do not want to reduce project scope, even if the Friday is going to be late, is, that we do not want to risk getting a lot of undone tasks ahead of us, as it might lead to a reduced project scope, which we are not interested in. Finally, allowing a developer to ask for a code review any time is a part of our method, to ensure high quality code.

6.2.4 Testing and Documentation

We have chosen to demand a high quality and thorough documentation of our code, due to several reasons. As described in the preface to this report, the developed code this semester is released as open source. It is likely that other development teams will further develop the code on future semesters or in a commercial

context. Because of that, it is necessary to provide a good documentation of the project, helping future development teams to understand and use the code. Furthermore, in our group, we are developing a library for the other groups in the multi project to use. The purpose of the library is to provide an API, allowing a GIRAF application to store and retrieve administrative settings from the administration module of GIRAF. Because this library is to be used by other groups on this semester, good library documentation is necessary (see Section 2.3 for a more detailed description of how the other groups are intended to use our library).

High quality code is a requirement for this semester, and while documentation is necessary for code to be considered high quality, code stability is also an issue. That is why we chose to deploy several methods for testing our software. Furthermore, a project related course Test and Verification (Test og Verifikation (TOV)) was given to us during the semester, so it has been a specific requirement that we use structured testing during the project.

System Architecture

7

In this chapter we describe the architecture of the administration module. The administration module consists of two elements: an administration module engine, and an administration module library. The engine is responsible for maintaining the integrity as well as providing access to GIRAF applications' administrative settings which are persistently stored in the database of the engine. Administrative settings are settings which are only intended to be accessible by a guardian, and it is the responsibility of our administration module to keep track of these settings, and ensure they also can be accessed and edited by applications including the library. Furthermore, the engine is responsible for providing a GUI on the Android device, such that guardians have a way of controlling the administrative settings of GIRAF and its installed applications. Besides handling administrative settings for GIRAF applications, the administration module is also responsible for handling the settings related to the GIRAF system itself. This includes settings related to the user of the GIRAF system, e.g. name, birthday, and a list of different abilities that describe what the user is capable of doing. By providing access for guardians to adjust these settings, the administration module can allow applications to customize their behavior according to the child's abilities.

7.1 Components

In Figure 7.1, an overview of the components in the system architecture is presented. At the bottom of the figure, the Android Package Manager is shown. The Android Package Manager is responsible for handling installation, upgrade, reset, and removal of packages on an Android device. Therefore, when an application is installed in GIRAF, the Android Package Manager will be handling all Android specific logic that is related to package handling on the device. For GIRAF, and especially the administration module, knowing what the Android Package Manager does, is of great interest. For instance, when a new package has been installed by the Android Package Manager, the engine of the administration module uses that information to read and store the administrative settings, which the newly installed GIRAF application has declared. A GIRAF application declares its administrative settings in an XML file named `settings.xml`, which must be bundled within the application before it is installed. The idea of representing administrative settings in XML is not a specific feature in Android. Instead it is unique to the administration module of GIRAF. As a note, the `settings.xml` file also allows an application to specify the contents of the administration GUI that is provided by the engine. This, and the concept of `settings.xml` are further described in Section 8.2.

For an application to be able to get and set administrative settings, it should include the administration module library. The library can be seen as an application's interface to the administration module engine, where the administrative settings are stored. The library utilizes a technique in Android designed for the particular purpose of allowing applications to communicate across the Android platform. The technique is known as a content provider and is described in detail in Section 10.1.

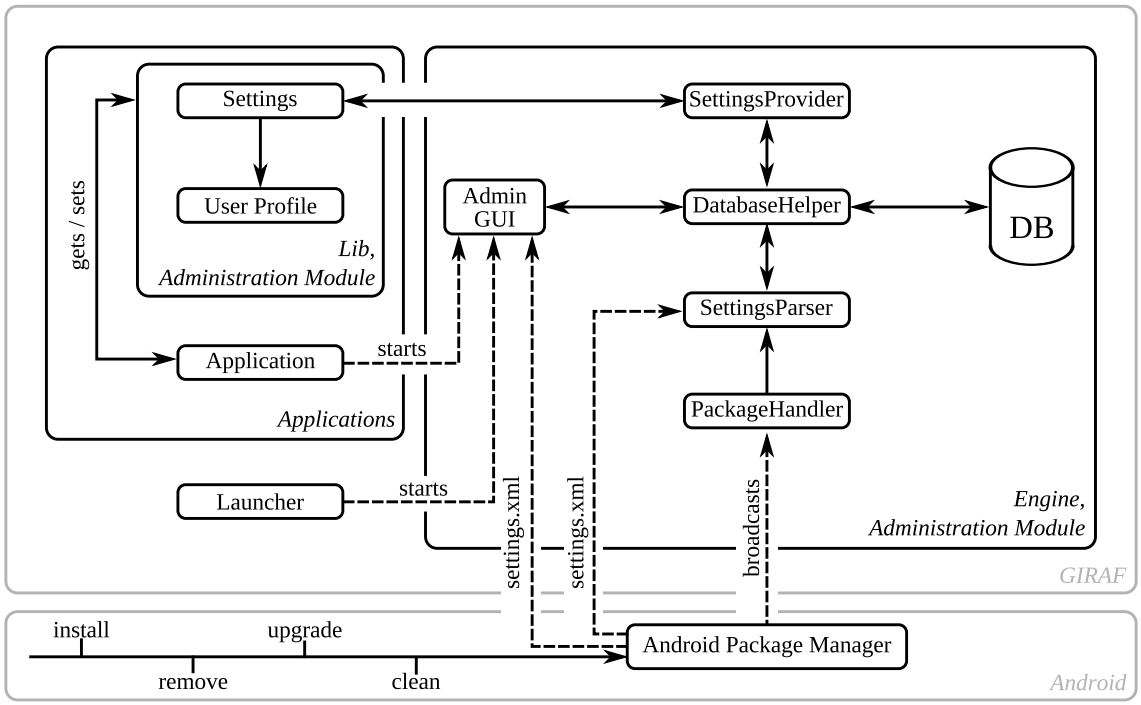


Figure 7.1. Shows the system architecture of the administration module in GIRAF, and how its components interacts.

In Android, a library cannot be installed separately on the device and be shared among different applications. This means, that each application will need to contain a copy of the library to be able to communicate with the engine. As covered in Chapter 9, the library offers an abstract view of the engine, enabling an application to get and set administrative settings by providing high level information to the implemented methods of the library. This means that all low-level technical aspects involved in getting and setting settings are handled within the library and engine implementation. In the engine, this is represented by the `SettingsProvider` which in technical terms is a content provider.

If we take a closer look at the engine, it consists of different elements. The `PackageHandler` is responsible for receiving notifications sent as broadcasts from the Android Package Manager. This could be notifications about a new package has been installed or that a package has been removed. In Section 10.4, the functionality of the package management in the engine, is covered in detail. In order to make it possible to handle the administrative settings of an application based on what the Android Package Manager broadcasts; the `PackageHandler` decides what needs to be done. For instance, if an application has just been installed, the `PackageHandler` will instruct the `SettingsParser` to parse and insert the administrative settings of the newly installed application into the database of the engine. In this process, the `SettingsParser` uses the database specific methods declared in the `DatabaseHelper` to insert settings into the database of the engine. The `DatabaseHelper` can be seen as an abstraction layer placed on top of the database, handling all the logic required when operating with a database on Android. How database handling is implemented in the engine, is covered in detail in Section 10.2 and Section 10.3. From the point where all the administrative settings of the newly installed application have been inserted, the `SettingsParser` has finished its job, and the administrative settings of the application are now persistently stored in the engine database.

From the launcher of GIRAF, a guardian can access the GUI of the administration module by entering a secret key combination. Doing that, will open the root menu of the administration module, making it possible for a guardian to adjust the administrative settings of each of the applications installed, as well as controlling settings related to GIRAF and the Android device. In Chapter 11, it is described how the GUI is generated and how it is presented to the user on the Android device.

Application Integration

8

In this chapter, we describe how an application is integrated with the administration module of GIRAF. The chapter is based on the demo application: BMI4Kidz, that has been briefly introduced in Chapter 4. Section 8.1 describes BMI4Kidz and its functionalities. In Section 8.2, it is described how an application can declare its administrative settings, such that the administration module of GIRAF is able to handle and represent them. Section 8.3 describes how BMI4Kidz represents its administrative settings to meet the functionality described in Section 8.1. As an application's administrative settings are defined in XML, and have to meet the syntax and semantics described in Section 8.2, a tool has been developed to ease the process of verifying if an application's settings meet the requirements. This tool is described in Section 8.4. In Section 8.5, it is covered how BMI4Kidz enables a guardian to edit the administrative settings of the application. The section covers both how BMI4Kidz can start its administration mode directly from within the application, and how the administrative settings can be accessed through the root menu of the administration module GUI in GIRAF. Finally, Section 8.6 describes how BMI4Kidz is using the library of the administration module to get and set some of the administrative settings that the administration module engine represents for the application.

8.1 BMI4Kidz

BMI4Kidz is a tiny application we have developed for two main purposes: to use it as a test application in the multi project test and to use it as an example-application in this chapter. Because of this, BMI4Kidz has been designed to be an application that uses the administrative functionality of GIRAF to a great extent. Basically, BMI4Kidz is a simple Body Mass Index (BMI) application, that uses a range of functionality of the administration module in GIRAF to compute and represent the BMI of the child using the GIRAF system. As BMI4Kidz is intended for small kids, the application allows a guardian to adjust the interval defining a normal BMI. In this way, the application can be customized specifically to the child using it. Furthermore, BMI4Kidz is also an application that allows guardians to control different settings ranging from basic height and weight adjustments to advanced functionality such as updating the profile image of the child using the BMI4Kidz application. And of course, because some people are very concerned about revealing their weight, BMI4Kidz also enables the guardian to manage the privacy settings of the application. Basically, the one and only privacy setting that BMI4Kidz provides, is to enable the guardian to toggle if BMI4Kidz should show the weight of the user, or not.

8.2 Administrative Settings

Administrative settings for an application can be seen as settings that the application developer intents only to be accessed by the guardians. An example of such an administrative setting could be defining whether or not the BMI4Kidz application should be allowed to show the user's weight. Another example could be settings allowing a guardian to adjust the interval for what is considered as a normal BMI. In general, an

administrative setting does not have to be related to privacy or security or something similar. Most of all, an administrative setting is declared to avoid a child from being able to control the setting - this could be for many reasons, and often it depends on the logic implemented in the particular application. Therefore, by declaring settings as administrative settings will mask them to the child, and will require the guardians to press a (secret) key combination before the administrative settings can be shown on the screen of the Android device. This also prevents the child from entering the administrative settings, and from getting lost in a variety of different properties and adjustments.

Declaring administrative settings are done using an XML file. We have defined that the XML file must be named `settings.xml` and that it must be located in the `/assets` folder of the particular APK having the settings. The `/assets` folder has been chosen as path for the `settings.xml` file because it allows other processes to get read-access to its files at runtime. In relation to the multi project, the `settings.xml` file can be seen as the link between an application and the administration module of GIRAF. In the following section, the syntax and semantics of `settings.xml` is defined.

8.2.1 Syntax and Semantics

The `settings.xml` file supports different levels of functionality, ranging from basic representation of primitives such as booleans and integers, to more advanced functionality, such as representation of objects that can be edited through activities made available on the Android device. In this section, the syntax and semantics of the `settings.xml` file will be described.

As `settings.xml` is based on XML, a Document Type Definition (DTD) has been made to define the valid document structure of the `settings.xml` file. The DTD of the `settings.xml` file supported by the administration module of GIRAF can be seen in Appendix C. In the DTD for `settings.xml` a range of different elements have been defined to allow an application to represent different types of administrative settings. Table 8.1 shows the data types of administrative settings that `settings.xml` allows an application to declare.

Primitives	**Objects**	**Enumerated**
`boolean`	`object`	`enum`
`integer`	`stdobject`	
`double`		
`string`		

Table 8.1. Shows the available data types in `settings.xml`.

All the primitives listed in the left column allow a developer to specify settings of these four types. Default values can be specified for all primitive settings, and if none are specified they will be computed in accordance with the rules listed in Appendix B. As an example, see line nine and ten in Listing 8.2. Here no default value has been specified for the height and weight of the child. According to the rules in Appendix B, this allows the administration module to initialize default values of these two settings to be as close to zero as possible. As another example, see line five in Listing 8.2 where **`true`** is explicitly set as default value for a setting of type `boolean`. A setting of type `object`, allows the application to store serializable objects. A setting of type `stdobject` can be specified to be one of the pre-defined types defined in the `sw6.lib.types` package. Common for settings of type `stdobject` is, that the GUI needed to enable the guardian to edit the object will be automatically generated by the administration module at runtime. The `enum` type allows an application to represent enums as an administrative setting.

As the administration module allows representation of an application's administrative settings when the GIRAF system has been switched into administration mode, the DTD of `settings.xml` has been designed such that it also allows representation of settings in a menu structure. In the DTD, this is accomplished by introducing a `menu` element containing the elements seen in Table 8.1. In the DTD, this is written as shown

in line four of Listing 8.1.

```
1 <!ELEMENT settings  (hidden?,visible?)>
2 <!ELEMENT hidden    (boolean|double|integer|string|object|stdobject|enum)*>
3 <!ELEMENT visible   (boolean|double|integer|string|object|stdobject|enum|menu)*>
4 <!ELEMENT menu      (boolean|double|integer|string|object|stdobject|enum|menu)+>
5 <!ELEMENT enum      (element)+>
```

Listing 8.1. Shows a snippet from the DTD of **`settings.xml`** showing how settings and menu structures must be declared.

As seen in Listing 8.1 line four, the DTD also allows a `menu` element to have a `menu` element as a child. Defining this allow us to provide a structure of the `settings.xml` file with support for sub-menus in as many levels as needed. The "+" sign following the `menu` element in Listing 8.1 declares that any of the allowed child-elements must occur one or more times inside the `menu` element. In this way, the DTD also defines that empty menus are not allowed. It should be noted that the DTD distinguishes between hidden and visible settings. This is seen in line one, where the DTD states that the `settings` element (which is also the root element of `settings.xml`), allows zero or one occurrence of a `hidden` element, followed by zero or one occurrence of a `visible` element. As seen in line two and three, the difference between the two elements is that a `visible` element allows declaration of `menu` elements whereas a `hidden` element does not allow this. In practice this means that all settings defined as `hidden` will not be visible through the administration module GUI in GIRAF, whereas all settings defined as `visible` will be visible in the administration module GUI. Instead, hidden settings will only be accessible through the library of the administration module. In Chapter 9, it will be described in detail how an application can retrieve settings stored in the administration module of GIRAF through the library.

Several attributes have been declared for the elements of `settings.xml`. In the following, all attributes will be briefly described. To make it easier to read, understand, and set the attributes into a "real" context, the `settings.xml` file of BMI4Kidz is listed below, see Listing 8.2. In Section 8.3 we describe the `settings.xml` file of BMI4Kidz in detail.

```
1  <?xml version = "1.0" encoding="UTF-8"?>
2  <!DOCTYPE settings PUBLIC "-//SW6B//DTD settings.xml//sw6.xmlvalidator" ↵
       "http://sw6android.lcdev.dk/sw6b.xmlvalidator/settings.dtd">
3  <settings>
4    <hidden>
5      <boolean varName="isFirstRun">true</boolean>
6      <integer varName="lastExecutedVersion"/>
7    </hidden>
8    <visible>
9      <double varName="height" realName="Height" desc="Height in centimeters (cm)" min="0"/>
10     <double varName="weight" realName="Weight" desc="Weight in kilograms (kg)" min="0"/>
11     <stdobject
12       varName="normalBmi"
13       realName="Interval for normal BMI"
14       desc="Adjust the interval for a normal BMI"
15       type="sw6.lib.types.Interval"/>
16     <object
17       varName="image"
18       realName="Image"
19       desc="Starts the camera to set the image of the user"
20       settingPc="null"
21       settingActivity="sw6.bmi.profile.EditImageActivity"
22       type="sw6.bmi.profile.Image" />
23     <menu title="Privacy" desc="Adjust visibility of personal settings">
24       <boolean
25         varName="showWeight"
26         realName="Show Weight"
27         desc="Define if the weight should be visible in the app.">true</boolean>
28     </menu>
29   </visible>
30 </settings>
```

Listing 8.2. The **`settings.xml`** file for BMI4Kidz.

varName
Required by: `boolean, double, integer, string, object, stdobject, enum`.
Represents the name of a setting. The name written here will be the identifier of the setting in the entire administration module of GIRAF, which is why the identifier has to be unique for all settings of the same type defined in the `settings.xml` file. Furthermore, it is required that the specified attribute value has a length equal to or longer than one character.

realName
Required by: `element`.
Optional in: `boolean, double, integer, string, object, stdobject, enum`.
If a setting is visible, this attribute will be the main text that identifies the setting to the user in the GUI of the administration module in GIRAF. For visible settings, this attribute is highly recommended to use. For an `element` in an `enum`, this attribute is required, as the attribute uniquely identifies elements in the enum.

desc
Optional in: `menu, enum, boolean, double, integer, string, object, stdobject`.
If a setting is visible, this attribute will be the text that describes the menu or setting in the GUI of the administration module in GIRAF. As with `realName`, this attribute is highly recommended for visible settings.

value
Required by: `element`.
As seen in line five in Listing 8.1, an `enum` contains one or more elements. For enum elements, identified as `element` in the DTD of `settings.xml`, the `value` attribute is required as it specifies the value of the `enum`. Like enums in the C programming language, `settings.xml` also allows the developer to explicitly specify the values of an enum's elements. The value is required to be specified, it must be unique inside the scope of a particular `enum`, and it must be specified as an integer.

title
Required by: `menu`.
The title of a `menu` is required to have a length of minimum one character. Multiple menus may have the same title. When a menu is declared, its title will be visible in the GUI of the administration module in GIRAF.

min, **max**
Optional in: `double, integer, string`.
For settings of type `double` and `integer`, the `min` and `max` attributes are used to define boundaries for the values that are assigned to these settings. For settings of type `double`, the `min` and `max` constraints are defined by values of type `double`. For settings of type `integer`, the `min` and `max` constraints are defined by values of type `integer`. For settings of type `string`, the `min` and `max` boundaries define the minimum and maximum allowed string length. For settings of type `string`, the length constraints are defined using natural numbers including zero.

settingActivity
Required by: `object`.
For settings of type `object`, the `settingActivity` attribute points to the activity that is started when a guardian edits the particular object. The activity must be implemented in the developer's application, and must take care of how the object is edited. The attribute is required, and it can only be specified for settings of type `object`. As the attribute points to an activity, the attribute's value must match the syntax of a class and its canonical class name, e.g. `sw6.bmi.profile.EditImageActivity`. As a remark, the `settingActivity` attribute does not need to point to an activity inside the application containing the `settings.xml` file. The attribute can also be used to start other activities, as long as an existing activity is pointed to.

settingPc
Required by: `object`.
For settings of type `object`, this attribute defines the class-path to a Java PC-GUI plug-in that could be loaded on a PC-interface when a guardian edits the object on a PC. As described in the future work chapter (see Section 14.1), in the beginning of the project, it was planned that a PC-Interface for the administration module should be developed as a part of this multi project. However, as time elapsed the idea was skipped as a lot of work still was to be done. At that point, the basic framework for the PC-Interface had been designed, and written into the `settings.xml` file. We decided to keep the attribute to make it easier for future developers on this project, to make the link between the administrative settings and a PC-Interface. Therefore, for the version of the administration module of GIRAF that is being developed this semester, this attribute is not supported and should be declared as **`null`**.

type
Required by: `object, stdobject`.
Must be set for settings of type `object` and `stdobject`. The type is required because we need a way to distinguish the different types of objects a developer can specify. As with `settingActivity`, the value of this attribute must match the syntax of a class and its canonical class name, e.g. `sw6.bmi.profile.Image`.

Based on the covered syntax and semantics of `settings.xml`, the following section will give an example of how `settings.xml` can be used to represent the administrative settings that BMI4Kidz should provide to the guardians of the GIRAF system.

8.3 Administrative Settings in BMI4Kidz

To represent the functionality of BMI4Kidz described in Section 8.1, the `settings.xml` file seen in Listing 8.2 has been made.

In BMI4Kidz, two hidden settings have been defined. In line five the setting `isFirstRun` is used to track the first time the application is started. BMI4Kidz uses this information to start the camera and let the user set the profile image shown in the application. In line six, `lastExecutedVersion` is declared as the second hidden setting. This setting is used to let BMI4Kidz track when it has been upgraded. When BMI4Kidz starts, it checks if the current version is newer than the version that was executed last time. If the current executing version of BMI4Kidz is newer, BMI4Kidz starts the camera, and lets the guardian update the profile image of the child using the application.

From line eight in Listing 8.2, the visible settings of BMI4Kidz are defined. The first two settings of type **`double`** declared in line nine and ten are used to represent the user's height and weight. As these settings are visible, they can be adjusted from the administration GUI in GIRAF. In line 11, a setting of type **`stdobject`** has been declared. The setting is of type `sw6.lib.types.Interval`, and it is used to enable the guardians to adjust the interval for a normal BMI. In line 16, an **`object`** setting has been defined. The object is used to store BMI4Kidz's profile image of the user. As seen in line 21, the `image` points to an activity of type `sw6.bmi.profile.EditImageActivity`. The activity starts the camera on the phone, and allows taking a new picture and storing it as the new profile image used in BMI4Kidz.

In line 23, a sub-menu has been declared. In this sub-menu, BMI4Kidz defines its privacy settings. The first and only privacy setting of BMI4Kidz is seen in line 24, where it is defined as a `boolean`. This setting allows the guardian to define if the child's weight should be visible in the application.

8.4 Validation Tool

As the administration module of GIRAF requires a valid `settings.xml` file to be bundled within the APK file of an application, it is very important that the `settings.xml` file has been validated according to the syntax and semantics covered in Section 8.2. The DTD is very useful for this particular purpose, as it describes the valid structure of `settings.xml`. However, even though a DTD allows independent groups of people to agree upon a standard DTD for the data specified in an XML file, there are limitations of a DTD that must be taken into account [W3S11]. For instance it is not possible to do type checking or implement constraints for attributes, e.g. by defining that a value of a `min` attribute always should be less than or equal to what might be defined in a `max` attribute. To ease the process of checking if a `settings.xml` file is valid according to the specified syntax and semantics in Section 8.2, we decided to develop a validation tool named `sw6.xmlvalidator`. Basically, the tool takes a `settings.xml` file as command line argument, and using the DTD it first verifies if the structure of the `settings.xml` file is correct. If the structure is valid, the tool continues to check if the semantics covered in Section 8.2 has been followed correctly. As the tool finds warnings and errors in the `settings.xml` file, it stores them in an internal data structure, and when the tool has finished checking the `settings.xml` file, it reports the results to the screen. The tool distinguishes between warnings and errors in the following way. A warning is given for several reasons, for instance if a visible setting has not been given a `realName`. A warning is not critical, and it will not block the particular `settings.xml` file from being used in the administration module of GIRAF. An error is critical, and will be reported if, for instance, two elements of the same type have the same name. If this is the case, the administration module will not be able insert the settings, as the database of the administration module does not allow settings of the same type and application having the same name. An error is blocking, and has to be fixed.

As developers are humans, they might forget to even run the validation tool. Therefore, the validation tool has been developed to support both desktop and server use. At the marketplace, named "GIRAF Place", the tool is used to verify if the APK file being uploaded has a valid `settings.xml` file. If errors are reported in this progress, the developer will be prevented from uploading the non-compatible GIRAF application.

The `sw6.xmlvalidator` has been written in Java to support as many platforms as possible, and it has been available for download from the sw6android Google Code project, since the first stable version was build. To help our fellow students to start using the tool, the documentation seen in Appendix D.8 was written. The documentation covers an introduction to the tool, its functionalities, and typical mistakes made when using the tool.

To give an illustration of the tool in progress, a couple of mistakes have been introduced in the `settings.xml` file for BMI4Kidz. In Figure 8.1, the `sw6.xmlvalidator` tool has checked the `settings.xml` file, and it reports the errors and warnings introduced. It should be noted that the `settings.xml` file seen in Listing 8.2 is valid, and does not contain any errors or warnings.

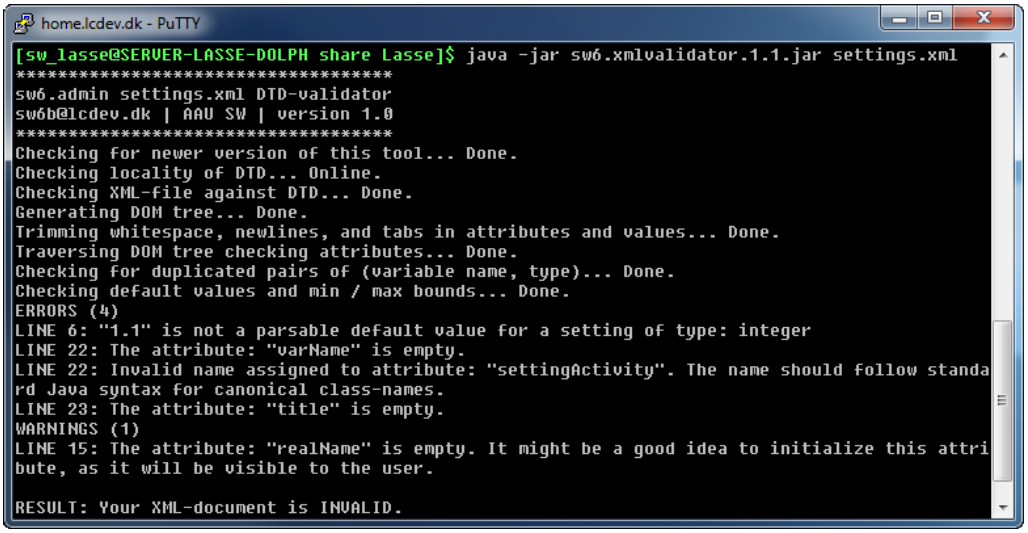


Figure 8.1. The result of validating a `settings.xml` file for BMI4Kidz that do not meet the syntax and semantics covered in Section 8.2.

As seen in Figure 8.1, the tool performs a sequence of different tasks that has not been covered in this section. For instance, the tool also checks if a newer version of the tool is available. This functionality has been implemented to ensure that application developers always check their `settings.xml` files against the latest syntax and semantics defined. Especially in the beginning of the multi project, where the concept of `settings.xml` was still being developed, this functionality was useful. If an older version of the tool is used, it simply rejects to validate the `settings.xml` file, and this helped us ensure that the application developers always checked their `settings.xml` file against the latest syntax and semantics.

8.5 Administrative Settings User Interface

After an application has been installed on the GIRAF system, the administration module has automatically parsed the `settings.xml` file, and inserted the declared settings into the database of the administration module. If a guardian wants to edit the administrative settings, there are two different approaches to use.

The first approach requires the guardian to bring the Android device to its home screen, e.g. by pressing the "HOME" button on the device. From the home screen, the guardian must press the volume up and down buttons in the following sequence: up, down, up, down, up, and down. When the volume down button has been pressed for the third time, the launcher of GIRAF will grant the guardian access to the root menu of the administrative module GUI. In Figure 8.2, the root menu of the administration module is shown. From here, the guardian can edit administrative settings for the device and the user profile, as well as download new applications or edit settings of applications like BMI4Kidz. If the guardian selects to edit the settings of BMI4Kidz, the screen in Figure 8.3, is shown.

The second approach is using the "in-app" administration mode. If a guardian wants to access the administrative settings for the current application running on the GIRAF system, the guardian can, from within the application, press the same key combination as covered before. Doing this will open the administration module of GIRAF, and show the administrative settings related to the particular application from where the administration module was opened. This is seen in Figure 8.3 where the root menu of the administrative settings for BMI4Kidz is shown. It should be noted that using the "in-app" administration mode is an optional feature that the particular application has to implement[1].

[1]The launcher group (sw6c), who is responsible for designing the code, that should enable any application to enter "in-app" admin-mode, has designed this feature such that it depends on each application to remember enabling the "in-app" admin mode.

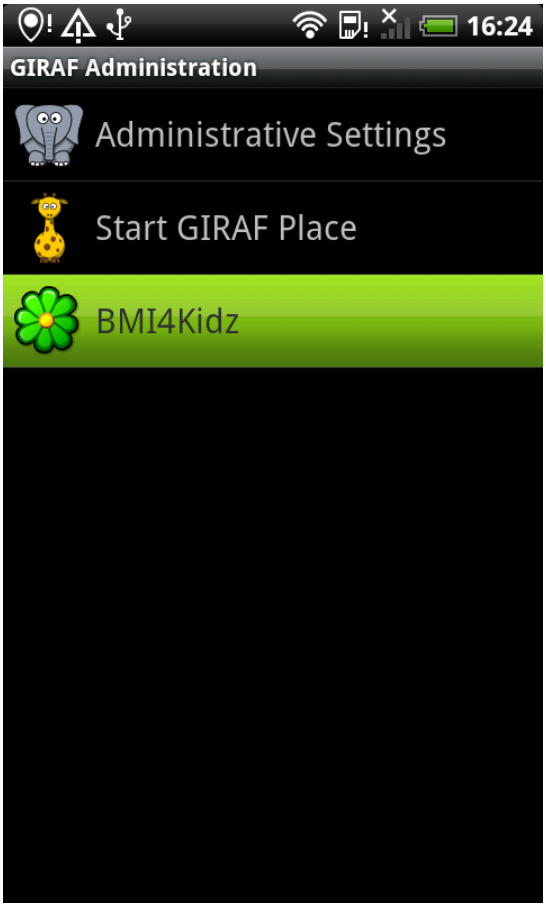


Figure 8.2. The root menu of the administration module in GIRAF.

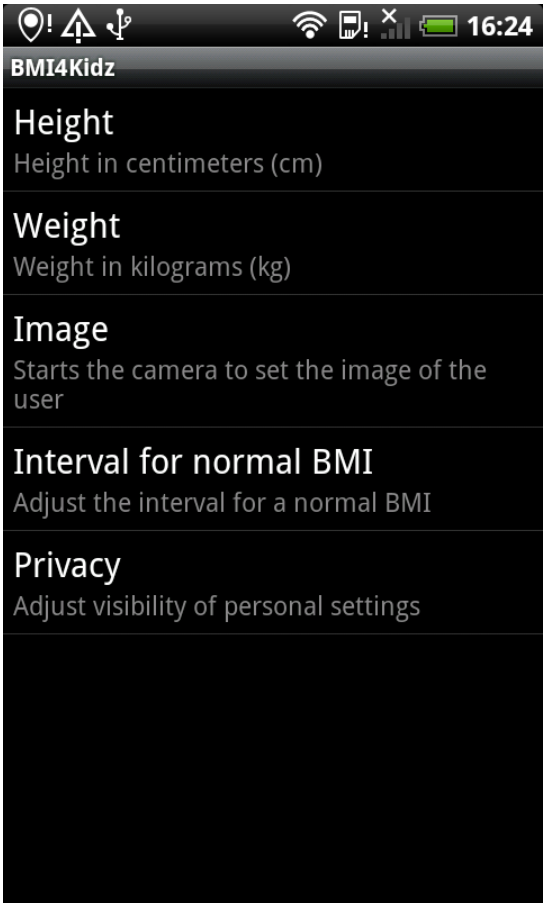


Figure 8.3. The root menu of the administrative settings available for the BMI4Kidz application. Here, all the visible administrative settings for the application can be edited.

8.6 Administrative Settings Library

As described in Section 8.5, after an application has been installed on the GIRAF system, the administration module has automatically parsed the `settings.xml` file, and inserted the declared settings into the database of the administration module engine. This section will describe how an application can get access to these settings. Again, the BMI4Kidz application will be used as example.

8.6.1 Application Specific Settings

For an application to get and set (read: update) the settings stored in the database of the administration module, the administration module library must be included in the application. Using the `sw6.lib.Settings` class, an application can get and set the settings at runtime. In BMI4Kidz, the hidden first run flag, the last executed version identifier as well as the height, weight, image, normal BMI interval, and the privacy setting for the visibility of the user's weight are retrieved through the administration module library. In Chapter 9, more details about the functionality provided by the administration module library are covered.

In Listing 8.3, it is shown how the BMI4Kidz application retrieves some of its settings from the administration module engine using the administration module library. In line two, three, and four, constants are declared to identify the settings. The constants are initialized with names matching the `varName` attribute

used for each of the settings declared in `settings.xml`. In line seven, eight, and nine, static methods in the `Settings` class are used to retrieve the values of the settings. The methods all take a parameter defining which application is asking for the setting and a parameter that identifies the setting - here the constants defined before are used. In line nine, it should be noted that the argument `Interval.class` is used to enable the library of the administration module to automatically cast and return the stored BMI-interval as an instance of the `sw6.lib.types.Interval` class. Finally, in line 12, the BMI4Kidz application use the values of the height and weight settings to compute the BMI of user.

```
1  // Constants matching the varName of the settings declared in settings.xml
2  private static final String sSETTING_DOUBLE_HEIGHT        = "height";
3  private static final String sSETTING_DOUBLE_WEIGHT        = "weight";
4  private static final String sSETTING_INTERVAL_NORMAL_BMI  = "normalBmi";
5  ...
6  // Get app settings from sw6.admin
7  double height = (Settings.getDouble(this, sSETTING_DOUBLE_HEIGHT)/100);
8  double weight = Settings.getDouble(this, sSETTING_DOUBLE_WEIGHT);
9  Interval normalBmi = Settings.getStdObject(this, sSETTING_INTERVAL_NORMAL_BMI, ↵
       Interval.class);
10 ...
11 // Compute BMI
12 double bmi = computeBmi(height, weight);
```

Listing 8.3. An example of how BMI4Kidz uses the administration module library to read the settings that have been declared in its **`settings.xml`** file seen in Listing 8.2.

8.6.2 User Profile

Besides allowing applications to define application-specific settings through a `settings.xml` file, a set of user profile settings are also available through the administration module library. As covered in Section 9.3, library provides information about the current user of the GIRAF system. Basically, the administration module library allows an application to get an instance of the `sw6.lib.user.Profile` class. An instance of the `Profile` class allows the application holding the instance, to retrieve the latest information about the user, e.g. name, gender, birthday, as well as different booleans defining the capabilities and handicaps of the user (can read? can hear? etc.). The `Profile` class and all of its encapsulated information are described in further detail in Section 9.3. In Listing 8.4, it is shown how BMI4Kidz retrieves an instance of the `Profile` class, and uses it to get information about the user. In line one, the instance is retrieved from the library. In line five the instance is used to get the name of the user, by writing: `userProfile.getName()`. Furthermore, in line five, using the name retrieved, a welcome message is printed to the screen. Finally, in line seven, the `Profile` instance is used to get the birthday of the user. The calendar returned is later used for printing the age of the user to the screen.

```
1  Profile userProfile = Settings.getUserProfile(this);
2  ...
3  // Set hello, welcome msg
4  TextView textViewWelcome = (TextView) findViewById(R.id.textViewWelcome);
5  textViewWelcome.setText("Hello, " + userProfile.getName() + " ;)");
6  ...
7  Calendar birthday = userProfile.getBirthDay();
```

Listing 8.4. An example of how BMI4Kidz uses the library of the administration module to read information about the user.

9 Library

The goal of this chapter is to describe the library of the administration module. First, the structure of the library is described in Section 9.1. Then, in Section 9.2, we describe how applications get and set administrative settings through the library and give examples of how this has been implemented. Following that, in Section 9.3, a description of how applications retrieve information about the user of GIRAF is given. Finally, in Section 9.4, the chapter describes how documentation of the library has been handled and an introduction to the Doxygen tool is given.

It should be noted that the library uses different technologies implemented in the administration engine. As the details of how the engine is implemented will be covered in the next chapter (Chapter 10), some of these technologies will only be covered briefly in this chapter. Furthermore, it should be noted that in Appendix D.5, the documentation of the library is given.

9.1 Structure

As covered in Section 7.1, the administration module library has the responsibility of providing an API for GIRAF applications. It is the goal of the library to allow applications to easily access administrative settings and user profile properties from the engine. This is illustrated in Figure 9.1. Using the library is optional, as the engine of the administration module can be communicated with, using a more low level functionality known as a content provider. The concept of a content provider and how it is implemented in the engine of the administration module is covered in Section 10.1. So, to serve as an abstraction layer above the low level communication opportunities provided by the engine, the library can be taken into use. When a GIRAF application includes the library, the code of the application and the code of the library will be joined at compile time, resulting in one Android `.apk` file being built. The file will then contain both the application and the library in one program. This is also illustrated in Figure 9.1.

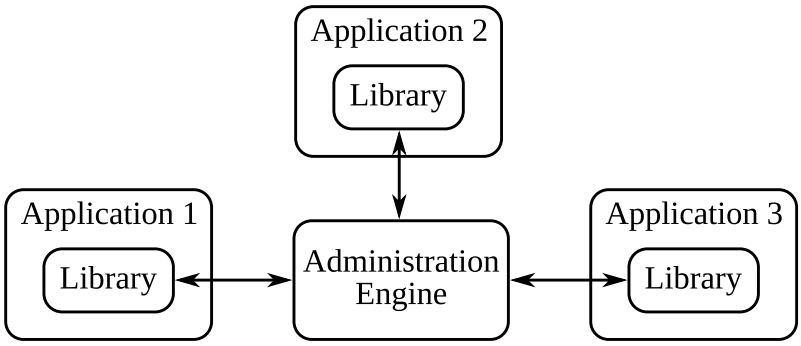


Figure 9.1. Shows how the library is used in applications to communicate with the administration module engine. The figure also shows the consequence of, that Android does not allow applications to depend on a shared library. As seen, this means that each application must include a copy of the library.

In Figure 9.1 it is illustrated exactly why the library might be useful to GIRAF applications. As mentioned briefly before, in order for a GIRAF application to communicate with the engine of the administration module, a content provider is needed because Android applications are not allowed to access each other's private data directly. If the library did not exist, GIRAF applications could still access application settings by communicating directly with the engine using the content provider. However, this would mean that applications developers would have to study how to use content providers, and spend a lot of time writing a system for using the content provider provided by the engine in the administration module. Instead the library provides an abstraction of all content provider usage, offering GIRAF applications a simple interface for communicating with the engine.

In addition to providing easy access to applications' administrative settings, the library also allows GIRAF applications to retrieve an instance of the `sw6.lib.user.Profile` class. Basically, the class contains information about the child using the device. The class will be further described in Section 9.3.

Besides the functionality described in this chapter, the library also contains classes belonging to group sw6c who is responsible for development of the launcher and the GIRAF Place. By allowing other groups to include files in our library, we make it easier for the application developers to integrate with GIRAF, as only one library then should be included to obtain all GIRAF relevant logic. This also eases integration across the entire platform.

9.2 Getting and Setting Settings

When an application wants to get or set the value of a specific administrative setting, this i done by calling a static method in the `sw6.lib.Settings` class. This class is the class in the library of "main interest" for the application developers, as it provides public methods to ease the communication with the content provider in the administration engine. Each type of setting has its own get and set method, e.g. `getInteger(...)` and `setInteger(...)`. This means, that if for instance an application has to set the value of a specific setting of type `string`, it will have to call the `setString(...)` method. The interface of `setString(...)` is seen in Listing 9.1.

```
1 public static void setString(Context context, String appName,
2   String varName, String varValue) throws SettingNotFoundException {
3   ...
4 }
```

Listing 9.1. Extract from the `sw6.lib.Settings` class showing the implementation of the `setString(...)` method.

The `setString(...)` method takes four parameters, as seen in Listing 9.1 line one and two. The `context` parameter is the instance of `android.content.Context` that belongs to the application trying to set the value of the setting. The `appName` parameter is the Android package name of the application which the setting belongs to. In the beginning, we were not aware of the fact, that the Android package name can be fetched from the instance of the `Context`, which is why there exist getter and setter methods with an arity of four, including the application package name. To introduce shorthands for getting and setting administrative settings through the library, we used method overload to declare a series of more "simple" getter and setter methods, which automatically fetches the application package name from the context, giving the methods an arity of three. We have not removed the "old" getter and setter methods, as they both allow GIRAF applications to get and set administrative settings for other applications, as long as the name of the other GIRAF application is known. For instance, if another application might be interested in the `weight` setting declared in BMI4Kidz, this is a fully possible and valid operation. The third parameter of Listing 9.1 is `varName` which is the name used to identify a specific setting. The `varValue` parameter is the new value of the setting.

Because the library does not have access to the engine database, it cannot directly update the new value of the setting. Instead, the administration module engine provides an Android content provider that can be used

to transfer the new value from the library to the engine - this is described in further detail in Section 10.1.

If an application needs to get the value of a specific administrative setting, the public getter methods defined in the `Settings` class can be used. Again, each setting type has its own get method. If an application needs to retrieve a specific setting of type `string`, it must call the static method `getString(...)`. The implementation of this method is shown in Listing 9.2.

```
1  public static String getString(Context context,
2    String appName, String varName) throws SettingNotFoundException {
3
4    Cursor settingCursor = getSetting(context, appName, varName, PrivateDefinitions.STRING);
5    String value = settingCursor.getString(0);
6    settingCursor.close();
7    return value;
8  }
```

Listing 9.2. The implementation of the `getString` method

In order to get the value of the setting, the private `getSetting(...)` method is first called in line four. This method utilizes the content provider of the administration module engine and returns an instance of the `android.database.Cursor` class. Basically, a `Cursor` encapsulates a result set returned by a database query. In Section 10.1 the details about how to use an instance of the `Cursor` class are covered in detail. Since a setting can only hold one value, the first value in the `Cursor` is retrieved. This value is fetched in line five, and returned to the application in line seven. In line six the database connections opened by the `Cursor` instance are closed.

9.2.1 Object Serialization

Not all methods for getting and setting administrative settings are this simple, and require more than just content provider communication. When a setting of type `object` or `stdobject` is to be set, the class instances first have to be serialized into a byte array. How this is done, is seen in Listing 9.3.

```
1  private static int setObject(Context context, String appName,
2    String varName, Serializable varValue) {
3    ...
4    ByteArrayOutputStream baos= new ByteArrayOutputStream();
5    ObjectOutputStream oos    = null;
6
7    try {
8      oos = new ObjectOutputStream(baos);
9      oos.writeObject(varValue);
10
11     byte[] byteArrayOutputSerialized = baos.toByteArray();
12     oos.close();
13
14     // Send byteArrayOutputSerialized to the engine through the content provider.
15     ...
16     // Return the number of rows affected by the update.
17     ...
18   } catch (IOException e) {
19     e.printStackTrace();
20     throw new SerializationException(
21       "The variable with the name: \"" + varName + "\" for the application: \"" + appName +
22       "\" could not be updated with the serialized value of the input object type: \"" +
23       varValue.getClass().getCanonicalName() + "\".");
24   }
25 }
```

Listing 9.3. Shows the implementation of the `setObject(...)` method declared in the `sw6.lib.Settings` class.

To serialize the new value of an object, the `java.io.ByteArrayOutputStream` class is instantiated in line four. In line eight, an instance of `java.io.ObjectOutputStream` is instantiated taking the `ByteArrayOutputStream` instance as parameter. The `ObjectOutputStream` is then used to serialize the `varValue` instance in line nine. The `ObjectOutputStream` will then write a series of bytes to the `ByteArrayOutputStream` representing the

class instance that is just being serialized. These bytes are then extracted in line 11 into a byte array. From that point, the byte array is encapsulated in a container object, used to represent the object while being sent via the content provider. All this "magic" starts from line 14. Please note, that details about how this actually works are covered in Section 10.1.

There is also a method for retrieving the value of settings of type `object` and `stdobject`. The implementation of this method is shown in Listing 9.4.

```
1  public static <T extends Serializable> T getObject(Context context,
2    String appName, String varName, Class<T> someClass) throws SettingNotFoundException {
3
4    // Request the engine for an object of the type specified by the generic class type.
5    // The byte array is returned encapsulated in a Cursor instance.
6    // Throw an exception if an object was not returned.
7    ...
8
9    // Fetch the byte array representation of the object.
10   byte[] byteArrayRestoredObject = settingCursor.getBlob(0);
11   ...
12
13   if (byteArrayRestoredObject.length == 0) {
14     return null;
15   }
16
17   ByteArrayInputStream byteArrayInputStream   = new ↵
         ByteArrayInputStream(byteArrayRestoredObject);
18   ObjectInputStream objectInputStream         = null;
19   Object object                               = null;
20   try {
21     objectInputStream   = new ObjectInputStream(byteArrayInputStream);
22     object              = objectInputStream.readObject();
23     objectInputStream.close();
24   } catch (Exception e) {
25   ...
26   throw new SerializationException(
27     "The variable with the name: \"" + varName + "\" for the application: \"" + appName +
28     "\" could not be loaded with the serialized value of the object type: \"" +
29     someClass.getClass().getCanonicalName() + "\".");
30   }
31
32   // Cast the object and return it
33   return someClass.cast(object);
34 }
```

Listing 9.4. Shows the implementation of the `getObject(...)` method declared in `sw6.lib.Settings`.

It should be noted that this method takes a class type as one of its parameters. This class type is for instance used to cast the de-serialized object into an instance of the correct class. The `getObject(...)` method works by first retrieving the object from the administration engine database. The object was stored as a byte array in Listing 9.3 and is also returned as a byte array. The byte array, representing the serialized class instance is fetched from the `Cursor` in line 10. In line 17, the `java.io.ByteArrayInputStream` class is instantiated taking the byte array as a parameter. In line 21, an instance of the `java.io.ObjectInputStream` class is created. This instance is used to de-serialize the byte array into an object of type `java.lang.Object`. In line 22, this is done by calling the `readObject()` method on the instance of the `ObjectInputStream`. Using the instance of the `ByteArrayInputStream`, the `ObjectInputStream` instance will read the byte array, and create the de-serialized object. In line 33, the de-serialized object is casted into a class specified by the `someClass` parameter passed as argument to the `getObject(...)` method in line 1. After casting the object, the instance is returned.

9.3 User Profile

The library provides a static, public method called `getUserProfile(...)` declared in the `sw6.lib.Settings` class. Applications can call this method to retrieve an instance of the `sw6.lib.user.Profile` class. The

`Profile` class provides access to all properties the administration module engine knows about the child using the GIRAF system. A first-run procedure, that is activated the first time the GIRAF device is started, prompts the guardian to set these settings. However, the guardian can skip this part, so there is no guarantee that the properties have been set, rendering them to contain default values. The main purpose of having these properties is to make it possible for GIRAF applications to compensate for physical or mental disorders the child might have, but also to use more generic information about the user, like name and gender. All the information that can be read from the `Profile` class is seen in Table 9.1.

Standard	**Can the user...**	**Does the user...**
Name	Drag and drop?	Require large buttons?
Birthday	Hear?	Have bad vision?
Gender	Understand analog time?	Require simple visual effects?
Address	Understand digital time?	Understand numbers?
Phone number	Read?	
	Speak?	
	Use a keyboard?	

Table 9.1. Properties of the user profile.

It is important to note that the `Profile` class does not contain the values for the properties listed in Table 9.1. Instead the `Profile` class serves as a lazy loader class that only loads the values when needed. If for instance, an application calls the `canRead()` method in the `Profile` class, a boolean value will be returned (false if the child cannot read or true if the child can read). No caching is implemented in the `Profile` class, so basically, the boolean value is loaded from the database every time `canRead()` is called. This ensures that an application always gets the current value of the attribute. An example of one of the lazy-loading methods can be seen in Listing 9.5.

```
1  public Calendar getBirthDay() {
2    int day       = Settings.getInteger(mContext,
3                    PrivateDefinitions.SW6_ADMIN_PACKAGE_NAME,
4                    PrivateDefinitions.PROFILE_BIRTH_DAY);
5
6    int month     = Settings.getInteger(mContext,
7                    PrivateDefinitions.SW6_ADMIN_PACKAGE_NAME,
8                    PrivateDefinitions.PROFILE_BIRTH_MONTH);
9
10   int year      = Settings.getInteger(mContext,
11                   PrivateDefinitions.SW6_ADMIN_PACKAGE_NAME,
12                   PrivateDefinitions.PROFILE_BIRTH_YEAR);
13
14   Calendar date = new GregorianCalendar(year, month-1, day);
15
16   return date;
17 }
```

Listing 9.5. Shows an extract from the `sw6.lib.user.Profile` class where `getBirthDay()` is implemented.

The `getBirthDay()` method in Listing 9.5 returns an instance of `java.util.Calendar` representing the birthday of the child. In line two, six, and ten, the birthday in form of three integers is loaded from the database using the `Settings` class in the library. These integers are used to create an instance of the `java.util.GregorianCalendar` class, seen in line 14. As a `GregorianCalendar` inherits from the `Calendar` class, the instance of the `GregorianCalendar` can be returned in line 16. Encapsulating the birthday date into an instance of the `GregorianCalendar` class, makes it easier for application developers to handle the birthday data, as a calendar object provides more information than three single integers.

With regards to performance of the `Profile` class, the response time of the get methods in the class would be better if the class had all the values pre-loaded. But the impact of this extra response time is very low, rendering it unimportant. Furthermore, the lazy-loading solution ensures that it is always the newest value of an attribute that is retrieved, since it is loaded directly from the database every time it is requested.

9.4 Library Documentation

Since the library is to be used by GIRAF application developers, thorough documentation on how to use the library is needed. All documentation is written in the form of Javadoc comments, a comment format that is natively supported in our MOTODEV IDE. Javadoc comments are placed right before a class or method definition for instance. An example of a Javadoc comment, documenting the method `getBoolean(...)` defined in `sw6.lib.Settings`, can be seen in Listing 9.6. The Javadoc comments are formatted by first having a general description of the method. After that, the parameters of the method are described, indicated by the "`@param`" tag seen in line seven, eight, and nine. Finally, the return value is described, followed by a description of the exceptions the method may throw, respectively denoted "`@return`" (in line 11) and "`@throws`" (in line 13).

```
1  /**
2  * Returns the requested variable as a boolean primitive.
3  * The variable is requested from the administration database.
4  * The method can also be used to get the value of a variable
5  * shared by multiple applications.
6  *
7  * @param context The context of your application.
8  * @param appName The name of some application.
9  * @param varName The name of the variable to request.
10 *
11 * @return The requested variable as a boolean primitive.
12 *         The variable is requested from the administration database.
13 * @throws SettingNotFoundException If the combination of application-
14 *         and variable name is not found in the administration database.
15 */
16 public static boolean getBoolean(Context context,
17   String appName, String varName) throws SettingNotFoundException {
18 ...
19 }
```

Listing 9.6. Shows a typical use of Javadoc comments.

In order to give an overview of the documentation and an easy way for GIRAF application developers to browse the documentation, a tool called Doxygen has been used. Doxygen is a tool for generating different types of documentation formats based on commented code. The tool support comments for a variety of different languages like C++, C, Java, Python, and PHP [vH11]. Using the Doxygen tool, we have generated HTML files representing documentation of all methods available in the `sw6.lib.Settings` class. The HTML files generated allow application developers to easily browse and view the documentation in a well-structured graphical way. The content of the Doxygen generated documentation is browsed using the tree view or the search function, both seen in Figure 9.2.

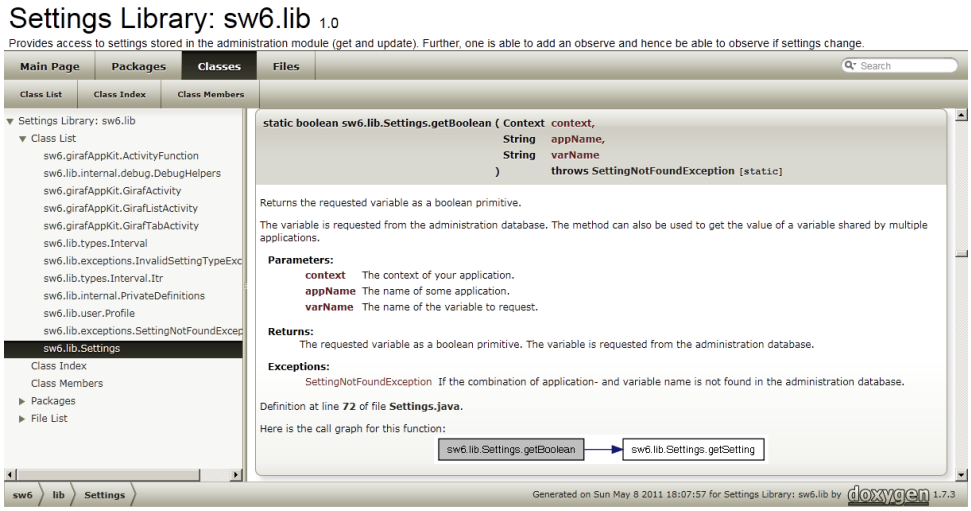


Figure 9.2. Shows a screenshot of the library documentation generated with the Doxygen tool.

In general Doxygen displays the documentation in a much more elegant way than the original Javadoc HTML files. A full copy of the Doxygen generated documentation for the library can be seen on the DVD in Appendix D.5.

It should be noted that in order to make it easy for our fellow students to always browse the latest version of the Doxygen generated documentation for the library, we used our own project site at `http://sw6b.lcdev.dk` to host the documentation. As we have accounted for earlier, an email was sent to the other students when new changes have been made.

Engine

10

In this chapter we describe the implementation of the engine in the administration module of GIRAF. First, in Section 10.1, we describe content providers and how this technology has been used to enable GIRAF applications to communicate with the engine of the administration module. Then, in Section 10.2 and Section 10.3 we describe the database of the engine. Here, we give an introduction to the embedded database engine in Android, and we explain how an abstraction layer above the database has been programmed to ease database communication as well as logic regarded to creation of the engine database. Finally, in Section 10.4, we describe how the administration module is able to insert, upgrade, and delete applications' administrative settings based on broadcasts sent from the Android Package Manager. Keywords like dependency injection and broadcast receivers are covered here also.

10.1 Content Provider

As we described in Section 5.1.3.4, the purpose of content providers is to enable communication between different applications across the Android platform. For the same reason, a content provider named `SettingsProvider` has been implemented in the administration module of GIRAF. The goal of the `SettingsProvider` is to provide an interface enabling communication between the administration module engine and the individual applications on the Android device. To make it easier for the application developers to communicate with the `SettingsProvider`, the library described in Chapter 9 has been developed. How the `SettingsProvider` integrates with the rest of the system is illustrated in Figure 10.1.

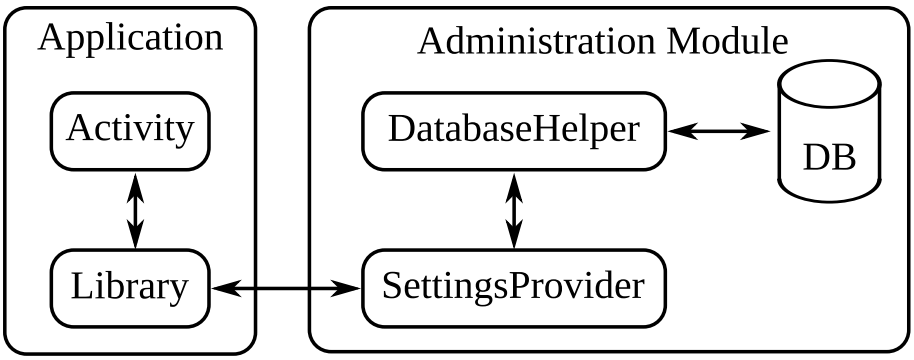


Figure 10.1. Shows the architecture of how the `SettingsProvider` is used as a link between an application and the database in the administration module engine.

Figure 10.1 can be divided into two parts: an application part and an administration module engine part. In the following two sections, we describe how the communication between these two parts is handled.

10.1.1 Implementation of SettingsProvider

Before a class can be considered to be a content provider on the Android device, the class first needs to extend the `android.content.ContentProvider` class and secondly, it must be declared as a content provider in the `AndroidManifest.xml` file. Besides declaring the canonical class-name of the class that will serve as a content provider, an identifier of the content provider must also be declared in the `AndroidManifest.xml` file [And11f]. It will be the identifier of the content provider that applications will use to get in contact with it. On the basis of what has been declared in the `AndroidManifest.xml` file, Android knows which class to invoke on a given request. Besides identifying the content provider, a request can also contain multiple parameters that can be used by the content provider. An example of a request sent to the `SettingsProvider` in the administration module of GIRAF can be seen in Figure 10.2. The example is based on the BMI4Kidz application we described in Chapter 8. In the example, the `weight` of type `double` is requested.

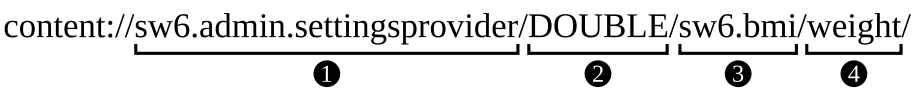


Figure 10.2. Shows how the structure of a content provider request can be divided into multiple elements. The structure seen here is used in `SettingsProvider` to allow applications to request any data type except for objects.

As seen in the example in Figure 10.2 a request is formed as an Uniform Resource Identifier (URI) and consists of multiple elements. Besides the first element (number one) identifying the content provider, the other elements are not controlled by Android and are therefore not required to be specified in the `AndroidManifest.xml` file. Instead it is the responsibility of the content provider itself to handle the other elements of a request. This can be compared to a web-server where the Uniform Resource Locator (URL) to this web-server is resolved by a Domain Name System (DNS) server. It is not the responsibility of the DNS server to decide which parameters are valid to be sent to the given web-server and how they should be interpreted. Instead, the DNS server makes sure that the given domain is resolved to a web-server Internet Protocol (IP)-address - and that is it. From that point, the web-server will receive all the parameters sent along with the URL, and it will be the responsibility of the web-server to decide if the request can be fulfilled or not. The different elements in Figure 10.2 have the following meaning:

1. Is the identifier of the content provider which Android will use to resolve it. This identifier must be declared in the `AndroidManifest.xml` file. For the `SettingsProvider` class, this identifier is set to: `sw6.admin.settingsprovider`.

2. This is the first parameter of the request. For the `SettingsProvider` this parameter indicates which type of setting that is requested. The type of a setting is a parameter, as we allow settings to have the same name, as long as their type is different- so to be able to distinguish between the different settings, we need the type. This is also the reason why we do not solely use the name of the setting to identify it. The different types that can be specified in this field are: `OBJECT`, `BOOLEAN`, `INTEGER`, `DOUBLE`, `STRING`, and `ENUM`. The types are equal to the types that can be specified in the `settings.xml` file, and are described further in Section 8.2.1. It should be noted that when working with settings of type `stdobject` the `OBJECT` type is used as the first parameter in the request.

3. For the `SettingsProvider` to be able to identify which application is requesting a setting, the package name of the application must be sent as a parameter of the request too. This is the second parameter of the request. The `SettingsProvider` does not check if the application asking for a setting is equal to the name of the application sent as the second parameter of the request. This means that every application on the GIRAF system can get access to each other's settings. This is not considered to be a bug, but instead as an opportunity for applications to share settings across the GIRAF platform, and in this way, affect each other's behavior. For instance, if an application might be interested in the height and weight of the user, the application could ask for these two settings by setting this second

parameter to point to the package name of BMI4Kidz which is `sw6.bmi`. Of course, this would involve that the application requesting this information has ensured that BMI4Kidz is already installed on the device.

4. The third and last parameter indicates the name of the setting that is requested.

As described in the caption for Figure 10.2, this structure is used to request all types of settings except settings of type `object`. For settings of type `object` it is necessary to add an additional parameter that defines the type of the `object` that is being requested. This is a required parameter as it is possible to define multiple objects having the same name, as long as their types are different. As a consequence, the name of the setting, and the fact that it is an `object` is not enough information to look up an `object` setting. An example of a request for an `object` setting is seen in Figure 10.3.

content://.../OBJECT/sw6.bmi/image/$\underbrace{\text{sw6.bmi.profile.Image}}_{\text{Type}}$

Figure 10.3. Shows an example of a request for a setting of type `object` belonging to BMI4Kidz. Besides the fact that the requested setting is of type `object`, information about the package name requesting the object, as well as information about the `object` name and type is required too.

To distinguish between valid and invalid URIs, the `SettingsProvider` uses an instance of the following class: `android.content.UriMatcher`. It is initialized as a constant named `URI_MATCHER`. The purpose of the `URI_MATCHER` is first of all to introduce it to which syntax is valid. From that point, all requests sent to the `SettingsProvider` will be sent directly to the `URI_MATCHER` which will evaluate if the request has a valid syntax or not. The code seen in Listing 10.1 defines the valid syntax for requests sent to the `SettingsProvider`.

```
1  static {
2    URI_MATCHER = new UriMatcher(UriMatcher.NO_MATCH);
3    URI_MATCHER.addURI(PrivateDefinitions.PROVIDER_CLASS,
4                       PrivateDefinitions.OBJECT  + "/*/*/*", MATCH_OBJECT);
5
6    URI_MATCHER.addURI(PrivateDefinitions.PROVIDER_CLASS,
7                       PrivateDefinitions.BOOLEAN + "/*/*",   MATCH_BOOLEAN);
8
9    URI_MATCHER.addURI(PrivateDefinitions.PROVIDER_CLASS,
10                      PrivateDefinitions.INTEGER + "/*/*",   MATCH_INTEGER);
11
12   URI_MATCHER.addURI(PrivateDefinitions.PROVIDER_CLASS,
13                      PrivateDefinitions.DOUBLE  + "/*/*",   MATCH_DOUBLE);
14
15   URI_MATCHER.addURI(PrivateDefinitions.PROVIDER_CLASS,
16                      PrivateDefinitions.STRING  + "/*/*",   MATCH_STRING);
17
18   URI_MATCHER.addURI(PrivateDefinitions.PROVIDER_CLASS,
19                      PrivateDefinitions.ENUM    + "/*/*",   MATCH_ENUM);
20 }
```

Listing 10.1. Shows an extract from the `sw6.admin.SettingsProvider` class. The `URI_MATCHER` is used to define which addresses can be received by the `SettingsProvider`.

In line two of Listing 10.1 an instance of the `UriMatcher` class is created. As default, the instance will reject any request regardless of its syntax. From line three to 18 a sequence of different syntaxes that the `UriMatcher` should accept, are added. This is done using the `addURI(...)` method, which takes three parameters defined as follows:

1. The first parameter defines the name of the content provider and is defined by using the constant `PrivateDefinitions.PROVIDER_CLASS` having the value: `sw6.admin.settingsprovider`. In Figure 10.2 it is seen that this parameter is the first element of the URI.

2. The second parameter is used to define which syntax is to be accepted by the `URI_MATCHER`. In line four of Listing 10.1 it is defined that the following syntax: "`OBJECT/*/*/*`" is a valid syntax. The "`*`" can be replaced by any symbols. In this way, it is declared that it is possible to send a request like the one shown in Figure 10.3, where the first parameter is `OBJECT` followed by three other parameters.

3. The last parameter is used to define an integer value for the syntax that is registered with the `URI_MATCHER`. By declaring this parameter it becomes possible for the `URI_MATCHER` to return a different integer dependent on which type of request it matches.

As we described earlier, before the `SettingsProvider` class is a content provider it needs to extend the `android.content.ContentProvider` class. And also, it needs to implement the following abstract methods defined in the `ContentProvider` class: `onCreate()`, `getType(...)`, `query(...)`, `insert(...)`, `update(...)`, and `delete(...)`. During development of the `SettingsProvider` and the administration module library, we decided not to offer GIRAF applications all the functionality that a content provider is capable of. The methods we decided not to implement are: `getType(...)`, `insert(...)`, and `delete(...)`. The reason for doing this is first of all, that the administrative settings an application provides must be declared in the `settings.xml` file. Hence, there will be no need for applications to dynamically insert or delete administrative settings at runtime. The logic that handles add and removal of administrative settings based on the `settings.xml` file is defined in the Package Handler of the administration module which is described further in Section 10.4. The reason why `getType(...)` is not defined, is because when an application asks for a setting, the application already know what type will be returned, as the type requested will be the type returned.

To show how the `SettingsProvider` communicates with the database of the administration module engine, the `query(...)` method will be explained in the following. The method is invoked when an application wants to get a setting through the `SettingsProvider`. Its implementation is seen in Listing 10.2.

```
1  public Cursor query(Uri uri, String[] projection, String selection,
2      String[] selectionArgs, String sortOrder) {
3    Cursor resultCursor = null;
4    int match = URI_MATCHER.match(uri);
5
6    if (match != -1) {
7      String appName = uri.getPathSegments().get(1);
8      String varName = uri.getPathSegments().get(2);
9
10     switch (match) {
11       case MATCH_ENUM:
12         resultCursor = dbHelper.getEnum(appName, varName);
13         break;
14       case MATCH_OBJECT:
15         String varCanonicalClassName = uri.getPathSegments().get(3);
16         resultCursor = dbHelper.getObject(appName, varName, varCanonicalClassName);
17         break;
18       ...
19       default:
20         return null;
21     }
22     ...
23   }
24   return resultCursor;
25 }
```

Listing 10.2. Shows an extract of the `query(...)` method declared in the `sw6.admin.SettingsProvider` class. In the extract the implementation of the `query(...)` method defines how requests for settings of type `enum` and `object` are handled.

The only parameter being used in the implementation of the `query(...)` method in the `SettingsProvider` is the one of type `Uri` seen in line one in Listing 10.2. The parameter contains a request, and is used as argument to the `URI_MATCHER.match(...)` method seen in line four. The result of running the `match(...)` method is an integer value denoting which of the syntaxes defined in Listing 10.1 that has been matched. If the URI represents a valid syntax, the application name and the variable name defined in the URI will be fetched in line seven and eight. Thanks to the way content providers are implemented in Android, this is straight forward as long as the parameters have been delimited by a slash "/", as seen in Listing 10.1. Dependent on which syntax that has been matched the switch case starting in line 10 will execute different branches. The switch case is used to determine the type of the setting and hence from which table the setting is to be selected. For instance if the URI requests a setting of type `object`, line 15 first gets the type of the `object` and then it queries the database for the setting.

As shown in Figure 10.1 in the beginning of this section, the `SettingsProvider` is communicating with the `DatabaseHelper`. An example of this is seen in line 16 in Listing 10.2 where the request of the `object` setting is forwarded to an instance of the `DatabaseHelper` class, identified as `dbHelper`. A description of the `DatabaseHelper` class is given in Section 10.3. The result of executing the `getObject(...)` method in line 16 is a `Cursor` that is returned in line 17 to the application which originally sent the request to the `SettingsProvider`. In the next section we will describe how to send a request to `SettingsProvider` from an application point of view as well as describe what a `Cursor` object is, and what it can be used to.

10.1.2 Use in Library

In the previous section we described how the content provider: `SettingsProvider`, is implemented in the engine of the administration module illustrated as the right hand side of Figure 10.1. This section will focus on the left hand side of Figure 10.1 and describe how the library is implemented to work with the `SettingsProvider`.

In theory it is possible for an application developer to use the `SettingsProvider` as any other content provider. However, it will require the developer to read the URI-syntax of the `SettingsProvider` thoroughly, and understand how it should be used and how its returned cursors should be handled. As the `SettingsProvider` class has been developed specifically to work with the engine of the administration module we assessed that it most likely would increase the risk of errors occurring if the developers should try to interface directly with the engine through the `SettingsProvider`. The purpose of the library is therefore to provide an interface that can be used to communicate with the `SettingsProvider` and at the same time, work as an abstraction layer on top of the logic needed to communicate with the `SettingsProvider`. In Listing 10.3 an example of a method implemented in the library is shown. The method is responsible for requesting all administrative settings except for objects.

```
1  private static Cursor getSetting(Context context, String appName,
2      String varName, String dataType) throws SettingNotFoundException {
3    Uri settingRequest   = Uri.withAppendedPath(
4                           PrivateDefinitions.CONTENT_URI,
5                           dataType + "/" + appName + "/" +  varName);
6    Cursor settingCursor = context.getContentResolver().query(settingRequest,
7                           null, null, null, null);
8
9    if(settingCursor == null || settingCursor.moveToFirst() == false) {
10     throw new SettingNotFoundException(appName, varName, dataType);
11   }
12
13   return settingCursor;
14 }
```

Listing 10.3. Shows an extract of the `sw6.lib.Settings` class where the `getSetting(...)` method is defined. The method is used to communicate with the `SettingsProvider` defined in the engine of the administration module. The method has its access modifier set to private as it is used as an internal method inside the `sw6.lib.Settings` class.

The first task of the method shown in Listing 10.3 is to create the URI that is used to contact and instruct the `SettingsProvider`. This is done in line three to five where an URI is created consisting of the constant `PrivateDefinitions.CONTENT_URI` (`sw6.admin.SettingsProvider`), and the parameters `appName` and `varName`. The last two variables have been passed to the `getSetting(...)` method as parameters. Due to the fact that `getSetting(...)` is a private method taken into use when querying settings of different types, a variable named `dataType` is also passed as parameter to the `getSetting(...)` method. In line four it can be seen that the class `PrivateDefinitions` is taken into use. This class consists of a number of constants that is used both by the library and the administration module engine. The class is defined in the library which is why the engine also includes a copy of the library. In this way, the address of the `SettingsProvider` is only declared one place, resulting in code that is easier to maintain.

In line six the context sent to the `getSetting(...)` method is used to request an instance of the `android.content.ContentResolver` class, which is used to communicate with content providers on the system. The newly created URI object from line three is then used in the query invoked on the `ContentResolver` instance and sent to the `SettingsProvider` in line six. The result of this request is an instance of the `Cursor` class returned by the `query(...)` method shown in Listing 10.2. In this way we have managed to invoke and execute a method that is running in the process of the engine, even though that it originally was invoked by an application running in another process, and this is the beauty of content providers. To be able to see the purpose of the instance of the `Cursor` class, it is necessary to look at a method that invokes `getSetting(...)`, see Listing 10.4.

```
1 public static int getEnum(Context context, String appName, String varName) throws ↵
      SettingNotFoundException {
2   Cursor settingCursor = getSetting(context, appName, varName, PrivateDefinitions.ENUM);
3   int value = settingCursor.getInt(0);
4   settingCursor.close();
5   return value;
6 }
```

Listing 10.4. Shows an extract of the `sw6.lib.Settings` class where `getEnum(...)` is implemented. As the method is a public method declared in the library it allows applications using the library to get a setting of type `enum` from the engine. The method uses `getSetting(...)` shown in Listing 10.3 to communicate with the `SettingsProvider`.

As seen in line two of Listing 10.4 the `getSetting(...)` method is invoked with parameters that have been described earlier, however, it should be noted that the last parameter defines the type of setting that is requested. The return value of the `getSetting(...)` method is an instance of the `Cursor` class containing the results of the request made. The result is stored inside this instance as a table with a pointer pointing to the current selected row inside the `Cursor` instance. The advantage of using a `Cursor` is that it can contain many different types of data. In the code seen in Listing 10.4 the instance: `settingCursor`, does only contain a single setting, as the `SettingsProvider` is designed to only return a single setting per request. This is seen in line three, where the first column in the internal table of the `settingCursor` is loaded and parsed as a setting of type `integer`. The difference between a setting of type `integer` and an `enum` in the engine is, that an `enum` is bounded to having only specified integer values declared in the `settings.xml` file of the application in question. Therefore, we recommend developers requesting and using settings of type `enum` to define constants (matching those in the `settings.xml` file) to be used to distinguish the semantics of the different values that an `enum` can be assigned. After the value has been fetched from the `settingCursor` it is very important to close the `Cursor` instance, allowing Android to free resources and close any database connections the `Cursor` might have opened. When this is done, the method can now safely return the value of the setting to the application that invoked the `getEnum(...)` method in the first place.

10.2 Database Structure

For persistent data storage, we have used the SQLite database engine embedded in Android. SQLite reads and write data to ordinary disk files, and allows a complete SQL database to be stored in a single file

[SQL11a]. SQLite is not a complete enterprise database designed to compete with MySQL or Oracle. Instead, SQLite aims at being small, fast, reliable, and a simple to use database engine that is easy to administer, operate, maintain, and embed into larger systems [SQL11b].

A transactional database is one in which all changes and queries appear to meet the Atomic, Consistent, Isolated, and Durable (ACID) criteria. SQLite is a transactional database, and basically this means that all changes made within a single transaction of SQLite either occur completely or nor at all. Even if the database engine, in the process of writing data to a database file, is interrupted by an application crash, an OS crash, or by a power failure, SQLite always guarantees the database to be consistent [SQL11e].

There are different versions of the SQLite database engine. As covered in Section 5.1.2.1, Android 2.2 comes with SQLite 3.6.22 [Car11]. Unlike SQLite 3.5.9 that was embedded in Android 2.1, the 3.6.22 version of SQLite adds support for foreign key constraints.

To enable storage of the administrative settings for applications installed in the GIRAF system, we designed the database schema seen in Figure 10.4. The schema-notation is based on Crow's foot notation described in [JB11]. Basically, the design is structured such that a table exists for each different data type that can be declared in the `settings.xml` file described in Section 8.2. In the tables storing primitive types of settings, a composite key consisting of the two attributes `app_name` and `var_name` is added in order to uniquely identify a setting for a particular application[1]. A composite key is a primary key composed of multiple columns creating a unique key [Bei07, P 322]. A primary key is a column in a table that makes each record unique [Bei07, P 178]. From the application developers' point of view, the composite key introduces a rule saying that an application must specify unique settings for each different primitive data type.

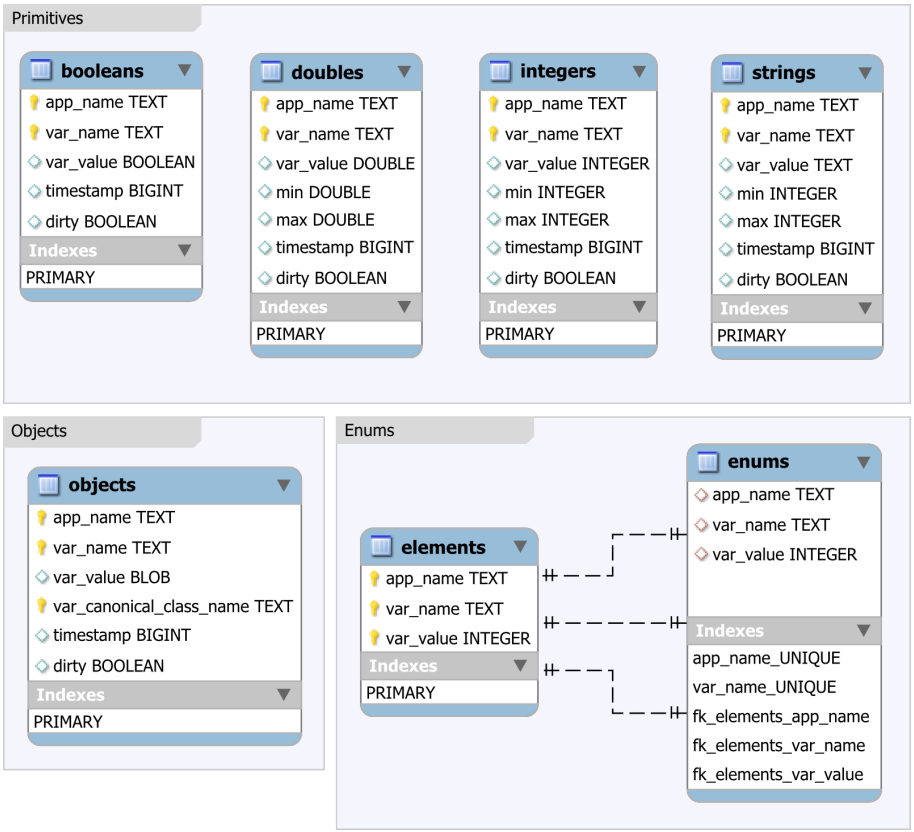


Figure 10.4. Shows the schema of the database used in the administration module engine of GIRAF.

[1]It should be noted that we later in the development phase realized that we could have optimized the database layout. In Chapter 14, Future Work, we will describe how a table, mapping an unique id to each `app_name`, could have been used to create a one-to-many relationship and avoid repetition of the application name inside the `app_name` attribute in all tables.

In Listing 10.5 the Data Definition Language (DDL) statement used to create the `doubles` table is seen. In line nine, the composite key is defined.

```
1  CREATE TABLE doubles (
2      app_name  TEXT,
3      var_name  TEXT,
4      var_value REAL,
5      min       REAL,
6      max       REAL,
7      timestamp BIGINT,
8      dirty     BOOLEAN,
9      CONSTRAINT 'doubles_primary_key' PRIMARY KEY ( app_name, var_name ),
10     CONSTRAINT 'min_max_constraint_check' CHECK ( min <= max
11             AND
12             var_value >= min
13             AND
14             var_value <= max )
15 );
```

Listing 10.5. DDL statement for creation of table `doubles`.

For the types `double`, `integer`, and `string`, the `min` and `max` attributes has been applied a check constraint to ensure that data of these three types can only be stored if it meets the minimum and maximum constraints. The check constraint is seen in line 10 through 14, and it states that `min` must always be less than or equal to `max`, and that `var_value` must always be greater than or equal to `min` and less than or equal to `max`.

The attributes `timestamp` and `dirty` were introduced in the beginning of the project, when the PC-Interface was still a part of our project scope. The attributes were intended to be used when settings should be synchronized with a PC-Interface. The `timestamp` indicates when a setting was recently updated, and the `dirty` flag indicates if a setting has been updated since last synchronization. As accounted for in Section 8.2, the PC-Interface was later removed from the project scope due to heavy workload. However, as GIRAF is intended to be a project suitable for future development, we decided to keep the attributes, as they easily enable database-layer support for implementation of data-synchronization with a PC-Interface.

Table `objects` is a bit different than the tables storing primitive settings. The main difference is the absence of the `min` and `max` attributes, and changes in the composite key. For administrative settings of type `object` we allow a developer to store as many objects as possible for a specific application, as long as the combination of application name, setting name, and setting type is unique. Basically this means, that multiple settings of type `object` are allowed to have the same name, as long as their types are different. In the `objects` table, the attribute `var_canonical_class_name` is used to identify the type of the object being stored, e.g. `sw6.lib.types.Interval`.

To store enumerated types, we introduced two tables: `enums` and `elements` as well as the use of foreign keys. For backward compatibility concerns across different SQLite versions, foreign key support must be explicitly enabled by executing the command: `PRAGMA foreign_keys = ON;` at runtime [SQL11d]. After doing this, foreign keys are working as intended. As seen in the database schema in Figure 10.4, the two tables `enums` and `elements` have a one-to-one relationship, meaning that an enum can only reference a single element in the `elements` table. The `elements` table stores all possible elements of a particular enum. In this table, the three attributes `app_name`, `var_name`, and `var_value` makes up the composite key, requiring unique enum elements to be stored for each enum. In the `enums` table, see Listing 10.6 line seven, the `app_name`, `var_name`, and `var_value` attributes are defined as a composite foreign key, each referencing a triple of attributes of the same name in the `elements` table. In this way, only enums with present and valid elements can be stored. In the `enums` table, a unique constraint has been added to the `app_name` and `var_name` attributes. By adding this constraint we ensure that the definition of an enum is met, as it disallows applications to store enums pointing to more than one element. Furthermore, cascades has been added to the `enums` table ensuring, that if the element, an enum points to is deleted or updated, the referencing enum tuple will also be updated or deleted too. The use of cascades is seen in line eight and nine.

```
1  CREATE TABLE enums (
2      app_name  TEXT,
3      var_name  TEXT,
4      var_value INT,
5      timestamp BIGINT,
6      dirty     BOOLEAN,
7      FOREIGN KEY ( app_name, var_name, var_value ) REFERENCES elements ( app_name, ↵
           var_name, var_value )
8        ON DELETE CASCADE
9        ON UPDATE CASCADE,
10     UNIQUE ( app_name, var_name )
11 );
```

Listing 10.6. DDL statement for creation of table `enums`.

10.3 Database Helper

In order to use the SQLite database engine library embedded in Android, we used the following class: `android.database.sqlite.SQLiteOpenHelper`, which is a helper class that can be used to manage database creation and version management of SQLite databases in Android [And11j]. As the class is abstract it needs to be extended by a sub-class. In the engine of the administration module we let `sw6.admin.database.DatabaseHelper` extend and implement the abstract methods of `SQLiteOpenHelper`. The abstract methods of the `SQLiteOpenHelper` class are:

`onCreate(SQLiteDatabase)`
This method is called when the database is created. Google Android recommends that table creation and data initialization takes place here [And11j]. The method takes an instance of the following class as parameter: `android.database.sqlite.SQLiteDatabase`. The instance references the newly created database.

`onUpgrade(SQLiteDatabase, int, int)`
When creating an instance of the `SQLiteOpenHelper` class, the constructor takes (among others) a database name and a version number as parameter. If a database of the same name already exists for the application in question, and if the version number has changed, the `SQLiteOpenHelper` makes sure that this `onUpgrade(...)` method is called. In this method, Google Android recommends that all logic regarding upgrade of the database takes place, e.g. by adding or removing tables to reflect the latest schema of the database [And11j]. Using the three parameters: the database, the old version number, and the new version number, the application upgrading the database, can determine what needs to be done.

`onOpen(SQLiteDatabase)`
This method is called when the database has been opened for read and write operations to be performed [And11j]. As parameter, the method takes an instance of the SQLite database that has been opened.

In the `DatabaseHelper` of the administration module engine, we have implemented the private method `onCreate(SQLiteDatabase)`, such that it handles everything needed when the database is created. First, we enable foreign key support by executing the SQL command: `PRAGMA foreign_keys=ON;`, required by SQLite [SQL11d]. Then, all tables described in the database schema in Figure 10.4 are created. Following that, we check if any GIRAF applications with a `settings.xml` file are installed on the system. If that is the case, we loop through each application and insert the settings specified in the applications' `settings.xml` file. In `onUpgrade(SQLiteDatabase, int, int)` and `onOpen(SQLiteDatabase)` we do not carry out any logic, besides from enabling foreign key support.

Besides taking care of creating, upgrading, and opening the database, we have developed the `DatabaseHelper` so that it also serves as an abstraction layer above the database. By doing this, we encapsulated all logic regarding insertion, upgrade, deletion, and selection of settings in one class, making it easier to maintain all database specific logic. In Listing 10.7, an example of how we have implemented selection of settings is shown.

```
1  public Cursor getDouble(String appName, String varName) {
2    return getSetting(appName, varName, TBL_DOUBLES);
3  }
4  ...
5  private Cursor getSetting(String appName, String varName, String table) {
6    Cursor settingCursor = this.getReadableDatabase().query(
7        table,                          // table to query
8        new String[] { COL_VAR_VALUE }, // value column
9        COL_APP_NAME + " = ? AND " +    // where clause
10       COL_VAR_NAME + " = ?",
11       new String[] {appName, varName},
12       null,                           // group by
13       null,                           // having
14       null);                          // order by
15
16   // Check if cursor contains any rows.
17   // If not, the request for a setting was formed incorrectly.
18   if(settingCursor.moveToFirst() == false) {
19     settingCursor.close();
20   }
21   return settingCursor;
22 }
```

Listing 10.7. Shows how selection of settings through the `DatabaseHelper` has been implemented.

In the particular example, line one offer a public method to retrieve a setting of type `double`. The method is used by the `SettingsProvider` described in Section 10.1. The method takes two parameters; the name of the application which setting is to be selected, and secondly, the name of the setting to select. As selecting settings of all types supported by the `settings.xml` file (except objects), uses the same database logic, all of this code has been extracted into a single `getSetting(...)` method starting from line five. Doing this minimizes code duplication and maintainability cost. The `getSetting(...)` method first gets an instance of the administration database with read permission, this is seen in line six. It should be noted, that if the database does not exists at this point, the call to `getReadableDatabase()` (implemented in the super-class: `android.database.sqlite.SQLiteOpenHelper`), will force `onCreate(SQLiteDatabase)` to be called in the `DatabaseHelper` class. The same procedure applies when `getWriteableDatabase()` is called. In this way, we are always ensured that a database exists before we ask the database for settings.

When a database exists, the `getReadableDatabase()` method returns an instance of the following class: `android.database.sqlite.SQLiteDatabase`. Using this instance, the SQL query defined in line six through fourteen can be executed. To see what really happens in this parameterized query, an example is given. For instance, if we want to select the *weight* setting declared in the `settings.xml` file for BMI4Kidz in Section 8.3, this will make the `query(...)` starting from line six execute the following SQL command:

```
select var_value from doubles where app_name = 'sw6.bmi'and var_name = 'weight';
```

As a result, an instance of `android.database.Cursor` is returned. The cursor provides read access to the result set returned by the query, and enables fetching of the result such that it can be used as the intended data type.

In Listing 10.8, another example of functionality provided by the `DatabaseHelper` is shown. Here, the method `updateSetting(...)` takes care of updating all settings, except settings defined as type `object`.

```
1  private static int updateSetting(SQLiteDatabase db, ContentValues contentValues, String ↵
       appName, String varName, String table) {
2    ...
3    int affectedRows = 0;
4
5    // If constraints are violated, an exception is thrown -
6    // we catch it here, and let the invoker handle error reporting.
7    try {
8      affectedRows = db.update(
9          table,
10         contentValues,
11         COL_APP_NAME + " = ? AND " +
12         COL_VAR_NAME + " = ?",
13         new String[] { appName, varName });
14   } catch(SQLiteConstraintException s) {
15     affectedRows = 0;
16   }
17   return affectedRows;
18 }
```

Listing 10.8. Shows how update of settings through the `DatabaseHelper` has been implemented.

In line eight, the update is carried out. The update method takes an instance of the following class: `android.content.ContentValues`, which contains a set of key-value pairs of data to be updated. The syntax of the parameterized query looks similar to the one used for inserts, and will not be further described. The update method returns the number of rows affected by the update. This number is returned to the invoker of the update, which will be responsible for taking the appropriate actions, e.g. throwing an exception, if zero rows were affected by the update. Furthermore, as seen in line 14, we catch the `android.database.sqlite.SQLiteConstraintException`. The exception is thrown by the SQLite library if we try to update a setting that violates the integrity constraints specified for the setting to be updated. For instance, the exception could be thrown if an application tries to set an integer to 3000, while it only allows values between 0 and 2000. If the exception is thrown, we handle it by setting the number of affected rows to zero, and let the invoker of the update handle error reporting. The "trick" by letting the invoker be responsible for exception handling is implemented, as it is not possible to throw exceptions via a content provider.

10.4 Package Handler

The purpose of the package handler is not to replace the functionality of the Android Package Manager described in Section 5.1.3.5. Instead, the package handler is intended to work as an extension having the responsibility of ensuring that all the administrative settings related to installed GIRAF applications are available in the database of the administration module. It is also the package handler's responsibility to ensure that the database only contains settings used by installed applications. If an application is deleted, or if an application removes settings during an upgrade, it is the package handler's responsibility to ensure, that the database is cleaned up, and do not contain any unused settings. The settings handled by the package handler, are the settings which an application specify in its `settings.xml` file. The `settings.xml` file is described in detail in Section 8.3. Figure 10.5 shows an overview of the package handler and the actions taking place outside the package handler.

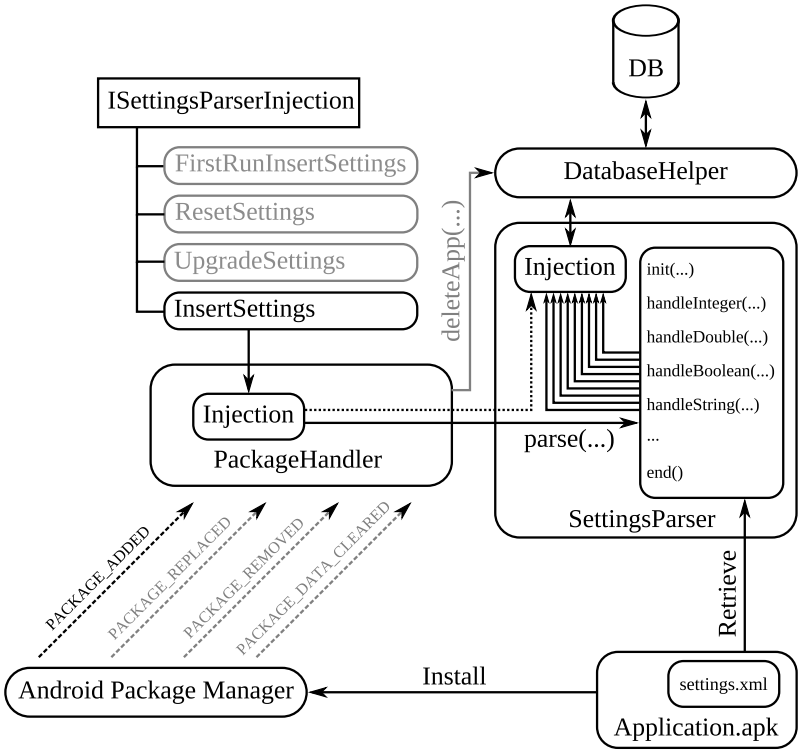


Figure 10.5. Shows the package handler and how it interacts with other elements of the system. Dotted lines from the Android Package Manager denote broadcasts that it sends. The figure indicates which modules are used during installation and insertion of a new GIRAF application. The gray elements are not used during this process, however, they are still a part of the logic that can be used by the package handler.

The package handling consists primarily of the `sw6.admin.PackageHandler` class. The `PackageHandler` class is a broadcast receiver and it reacts on broadcats sent from the Android Package Manager. Depending on which broadcast has been received, the `PackageHandler` creates an instance of a class that implements the `ISettingsParserInjection` interface. Basically, an instance of such a class dictates how settings should be handled. Different instances are created dependent on the broadcast received. Figure 10.5 is an example of the actions taking place when a package is being installed on an Android device with GIRAF installed. The following description will serve as a short explanation of the example.

The Android Package Manager is asked to install the package *Application.apk* on the Android device. When the installation has finished, the Android Package Manager sends the `PACKAGE_ADDED` broadcast. This broadcast is received by the `PackageHandler` class which reacts on the received broadcast. As the received broadcast is a `PACKAGE_ADDED` broadcast, the `sw6.admin.parser.SettingsParser` will be called from the `PackageHandler` along with an instance of the `sw6.admin.parser.InsertSettings` class. From that point, the `SettingsParser` will traverse through the `settings.xml` file bundled with the installed application, and using the logic specified in `InsertSettings`, each setting found in the `settings.xml` file, will be inserted into the database.

The next sections in this chapter describe the individual sub-components of the `PackageHandler` in detail, and focuses on the implementation. Besides covering installation of packages, the next sections will also consider how upgrade and removal of packages are handled.

10.4.1 Broadcast Receiver

Receipt of broadcasts is done in the `PackageHandler` class, extending `android.content.BroadcastReceiver`. Listening for broadcasts allows us to react when different actions have been carried out by the Android

Package Manager. To enable receipt of broadcasts sent from the Android Package Manager, Android needs to know that the `PackageHandler` class is listening. It is therefore necessary to specify in the `AndroidManifest.xml` file, that the `PackageHandler` class is a broadcast receiver, as well as which broadcasts it receives.

```
1 <receiver android:name="PackageHandler">
2   <intent-filter>
3     <action android:name="android.intent.action.PACKAGE_ADDED"/>
4     <action android:name="android.intent.action.PACKAGE_REPLACED"/>
5     <action android:name="android.intent.action.PACKAGE_REMOVED"/>
6     <action android:name="android.intent.action.PACKAGE_DATA_CLEARED"/>
7     <data android:scheme="package"/>
8   </intent-filter>
9 </receiver>
```

Listing 10.9. Shows an extract from the `AndroidManifest.xml` file defining which broadcasts can be sent to the `PackageHandler` class.

In line three to six of Listing 10.9, the four different broadcasts that can be received by the `PackageHandler` class are defined. In line seven it is defined that besides receiving broadcasts, the `PackageHandler` should also receive information about which package each broadcast is related to. For instance, this makes it possible to see which package has been installed on the Android device. As covered in Section 5.1.3.1, a broadcast is a special type of intent, which also explains why the type of broadcasts specified in line three through six, has the prefix `android.intent.*`, and not `android.broadcast.*` for instance.

When a broadcast related to the `PackageHandler` is sent, Android creates an instance of the `PackageHandler` class, and invoke its `onReceive` method. Parameters sent to the `onReceive` method are the context in which the broadcast receiver is running as well as the intent created and broadcasted by the Android Package Manager. After receiving the broadcast, it is now up to the `PackageHandler` to perform the actions dependent on the type of broadcast received. Before taking any action, the `PackageHandler` also performs different kinds of checks (will be described in the following). An extract from the `PackageHandler` class is seen in Listing 10.10 where it is assumed that the received broadcast is a `PACKAGE_ADDED` broadcast.

```
1  public void onReceive(Context context, Intent intent) {
2    String appName = intent.getData().getSchemeSpecificPart();
3
4    if (appName.startsWith(PrivateDefinitions.GIRAF_APP_PREFIX) &&
5        DatabaseHelper.checkDataBaseExsists(context)) {
6
7      if (intent.getAction().equals(Intent.ACTION_PACKAGE_ADDED)) {
8        if (!intent.getBooleanExtra(Intent.EXTRA_REPLACING, false)) {
9          // Call the parser and parse Settings.xml for the application
10 ...
```

Listing 10.10. Shows an extract from the `sw6.admin.PackageHandler` class where different checks are made after a `PACKAGE_ADDED` broadcast has been received.

The first information retrieved, is the name of the package that has led the Android Package Manager to send out the broadcast. This information is retrieved using the `getSchemeSpecificPart()` method, which retrieves the information specified in line seven of Listing 10.9, where the scheme specific part was set to: `<data android:scheme="package"/>`. The first if-statement, line four, contains two checks where the first tries to filter packages with the "`sw6.*`" prefix. Early in the multi project, it was decided that all GIRAF packages should have this prefix.

The second statement of the combined if-statement in line four is made to check if the database of the administration module has been created. As described in Section 10.3, the `onCreate(...)` method of the `DatabaseHelper` is called when the administration database is created. In this method, it is checked if any GIRAF applications are installed on the system, at the time the database is created. If this is the case, the settings of the installed GIRAF applications will be inserted into the administration database. So, because we handle insertion of applications' administrative settings when the database is created, we should not try

to insert settings in the PackageHandler if the database does not exists at the point where the broadcast is received.

Line seven and eight in Listing 10.10 checks the type of the broadcast received. If it is of type ACTION_PACKAGE_ADDED, it will be checked if this broadcast has been sent as a part of an upgrade of the package in question. The Android Package Manager is designed such that when a package is upgraded, the following three different intents are broadcasted: ACTION_PACKAGE_REMOVED, ACTION_PACKAGE_ADDED, and ACTION_PACKAGE_REPLACED. Furthermore, extra information is bundled with the ACTION_PACKAGE_REMOVED and the ACTION_PACKAGE_ADDED broadcast, which is identified by the name EXTRA_REPLACING. The purpose of this bundled information is to be able to tell if the received broadcast has been sent due to a package upgrade. For this reason, line eight of Listing 10.10 rejects to handle the broadcast, if it appears, that the broadcast received actually is intended to be handled as an upgrade. The same applies for broadcast received of type ACTION_PACKAGE_REMOVED. To handle package upgrades, the PackageHandler instead listens for ACTION_PACKAGE_REPLACED broadcasts. If such a broadcast is received, the PackageHandler knows for sure, that the package has been upgraded, and that the related logic to handle this should be executed.

If all checks in Listing 10.10 evaluates to true, the PackageHandler will handle the broadcast by invoking the SettingsParser.parse(...) method, and send the appropriate instructions of what needs to be done. What actually happens in the SettingsParser and its parse(...) method is described in detail in the following section. As an exception, the SettingsParser.parse(...) method will not be invoked if a package has been removed from the system. This case is described in Section 10.4.6.

10.4.2 Parsing Settings

Basically, the job of the SettingsParser.parse(...) method is to handle a settings.xml file by parsing its declared settings and store them in a temporary data structure. Depending on the type of broadcast received by the PackageHandler, the parser is instructed to carry out a specific action. For example, during installation of a GIRAF package, all settings declared in the settings.xml file of the particular package must be inserted directly into the database of the administration module. However, during upgrade of a package, the settings declared in the settings.xml file must be analyzed to check which settings to delete, insert, and update in the administration database.

In the beginning of the development phase, the SettingsParser was written to handle installation of new GIRAF packages. However, it quickly turned out that this functionality was not sufficient, as support for package removal, reset, and replacement also were to be added. The structure of the SettingsParser was therefore redesigned such that all settings first are parsed and then each type of setting is stored in its own list. For instance, all settings of type double are stored in a separate list and so forth. Then, all lists of settings are traversed, and the information regarding each setting is handled in the SettingsParser. One of the simplest types to handle is settings of type object, and an example of how this is done is seen in Listing 10.11.

```
1 ...
2 for(int i = 0; i < objectSettings.getLength(); i++) {
3   Element setting = (Element) objectSettings.item(i);
4   String varName  = setting.getAttribute(Definitions.XML_ATTRIBUTE_VAR_NAME);
5   String type     = setting.getAttribute(Definitions.XML_ATTRIBUTE_TYPE_NAME);
6
7   injection.handleObject(varName, type, PrivateDefinitions.DEFAULT_OBJECT);
8 }
9 ...
```

Listing 10.11. Shows an extract from sw6.admin.parser.SettingsParser where a for-loop traverses all objects defined in an applications settings.xml file.

In Listing 10.11 the code traversing all parsed settings of type object is shown. As defined in the DTD for settings.xml (see Appendix C), a setting of type object has, among other attributes, the two

attributes `varName` and `type` (see Section 8.2.1 for a detailed description of the attributes). In the for-loop of Listing 10.11, the goal is first to get the necessary information about each `object` setting, such that they can be stored in the administration database. For settings of type `object` we only need to know the name and type of the object, as well as the name of the application to which the setting belongs. This can be concluded by looking at the schema for the administration database shown in Figure 10.4. The first two pieces of information are retrieved in line four and five of Listing 10.11, whereas the application name is retrieved in the instance of the used injection. Using the information we got about a particular `object` setting, line seven calls the `handleObject` method on an instance of an injection, which take over the responsibility for what is going to happen next with the `object` setting. The `injection` variable references an object that was handed to the `SettingsParser.parse(...)` method at the time it was invoked. The object implements the `ISettingsParserInjection` interface, shown in Listing 10.12.

```
1  public interface ISettingsParserInjection {
2    public void init(Context context, String appName);
3
4    public void handleInteger(String varName, int min, int max, int varValue);
5    public void handleDouble(String varName, double min, double max, double varValue);
6    public void handleBoolean(String varName, boolean varValue);
7    public void handleString(String varName, int min, int max, String varValue);
8    public void handleObject(String varName, String type, byte[] varValue);
9    public void handleEnumElement(String varName, int varValue);
10   public void handleEnum(String varName, int varValue);
11
12   public void end();
13 }
```

Listing 10.12. Shows the definition of `sw6.admin.parser.ISettingsParserInjection` defining the interface that all injections must implement to be used with the `SettingsParser`.

As seen in Listing 10.12, methods to handle each type of setting that can be specified in the `settings.xml` file are defined. By programming different classes which implements this interface, it is possible to control how different types of a parsed `settings.xml` file should be handled dependent on the broadcast received by the `PackageHandler`. Besides the methods to handle each type of setting, two other methods are also defined: `init(...)` in line two, and `end()` in line 12. The purpose of defining those methods, is to allow the classes implementing this interface, to execute code before handling of settings starts, and after handling of settings has completed. For example, the methods could be used to open a database connection before the parsing starts, and keep it open until the parsing completes.

The principle of inserting an object that defines some of the behavior that a class like `SettingsParser` is to perform, is known as "Dependency Injection". Normally, dependency injection is used when a certain class is instantiated, and an object ("injection") implementing a common interface is sent as argument to the constructor of the class being instantiated. So, since the object injected into the `SettingParser`, is handed to the `SettingsParser` through the static `parse(...)` method, we do not exactly meet the specification of dependency injection as it has been described by Martin Fowler in [Fow04]. However, as we are very close to the definition, and as we adapt the concept to a great extent, we will still allow ourselves to call our use of injections, for dependency injection.

In the administration module we have written the following four different classes implementing the `ISettingsParserInjection` interface: `FirstRunInsertSettings`, `ResetSettings`, `UpgradeSettings`, and `InsertSettings`. These four classes can also be seen in Figure 10.5. The remaining sections of this chapter will describe the different implementations of the `ISettingsParserInjection` interface and which rules that have been defined at installation, upgrade, and reset of packages. The sections will therefore primarily focus on the communication carried out between each injection and the `DatabaseHelper` - a relationship which also is illustrated in Figure 10.5. As an exception, the last section will not consider an injection as it describes how removal of administrative settings is carried out when an application is being removed from GIRAF.

10.4.3 Installation

After a GIRAF package has been installed, the `PackageHandler` creates an instance of the `InsertSettings` class, and sends it as parameter to `SettingsParser.parse(...)`. During installation of a new package there is no need to take existing settings into account, as they simply do not exists for the package being installed. For this reason, `InsertSettings` is a very simple injection. Listing 10.13 shows an extract of the injection, where all settings of type `integer`, defined in the `settings.xml` file, are inserted directly into the database of the administration module.

```
1  ...
2  public void init(Context context, String appName) {
3    this.db = new DatabaseHelper(context);
4    this.appName = appName;
5  }
6  public void handleInteger(String varName, int min, int max, int varValue) {
7    db.insertInteger(appName, varName, varValue, min, max);
8  }
9  ...
10 public void end() {
11   db.close();
12 }
```

Listing 10.13. Shows an extract of `sw6.admin.parser.InsertSettings` used when administrative settings are inserted into the database of the administration module.

In the `init(...)` method seen in line two, the name of the application being installed is stored in a private variable. Furthermore, an instance of the `DatabaseHelper` is created. For every setting of type `integer`, the `SettingsParser` has parsed from its current `settings.xml` file, the `handleInteger(...)` method seen in line six is invoked on the injection. As seen in line six, the information handed to `handleInteger(...)` is then directly handed to the database helper, where the `insertInteger(...)` method in line seven, is used to insert the parsed setting into the database of the administration module. When no more settings needs to be handled by the `SettingsParser`, the parser will invoke the `end()` method on the `InsertSettings` instance, which will then close its connection to the database.

10.4.4 Clear Settings

Reset of settings for an application is implemented in a way that resembles how insertion of settings is implemented. The main difference between the two cases is that when clearing settings for an application, the old settings will already be present in the administration database. Therefore, the data needs to be overwritten with the default values declared in the `settings.xml` file. To accomplish this, an instance of the `ResetSettings` injection is created, and passed to the `SettingsParser` as argument to its `parse(...)` method. An extract of the `ResetSettings` injection is seen in Listing 10.14.

```
1  ...
2  public void handleInteger(String varName, int min, int max, int varValue) {
3    db.updateInteger(appName, varName, varValue);
4  }
5  ...
```

Listing 10.14. Shows an extract of `sw6.admin.parser.ResetSettings` used when administrative settings are cleared and reset to default values in the administration database.

Unlike the installation procedure where settings were inserted directly into the administration database, the settings are now just reset to their default values, specified by the `varValue` argument handed to `handleInteger(...)` in line two. The reset of integers is carried out by the `updateInteger(...)` method seen in line three. The method is invoked on an instance of the `sw6.admin.database.DatabaseHelper`, and updates the setting to the value specified by the `varValue` variable which, in this context, contains the default value of the setting.

10.4.5 Upgrade

Upgrade of packages in GIRAF is the most challenging case to support in the package handling process. The upgrade of administrative settings is handled by an `UpgradeSettings` injection, which also implements the `ISettingsParserInjection` interface. The main challenge is to solve conflicts and decide which settings to keep and which to delete. During development of the administration engine, we decided that upgrade of settings should be implemented such that it preserves as many settings as possible in the database. Doing this, should make it easier for the application developer to know, what is actually going to happen on package upgrades. The consequence of this decision is that `UpgradeSettings` has to take many different cases into account, define rules for all of them, and then ensure that all of the rules are correctly implemented in the class. An overview of the `UpgradeSettings` injection is given in Figure 10.6. The figure shows how the `init(...)` and `end()` methods are used, which is indeed different from the more simple cases, e.g. the logic implemented in the `InsertSettings` injection.

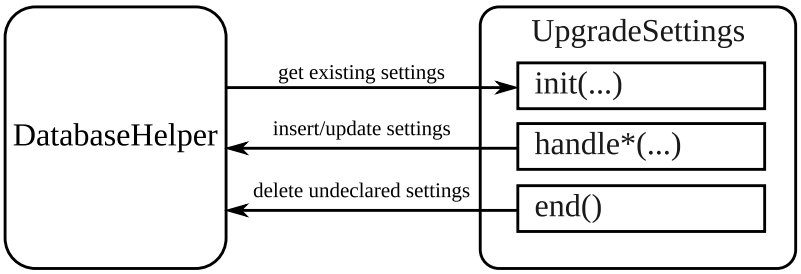


Figure 10.6. Shows the communication between the `UpgradeSettings` injection and the `DatabaseHelper` when the administrative settings of a package are being upgraded.

When the `init(...)` method is called, a series of different internal lists are created in the `UpgradeSettings` injection. The lists contain application settings already stored in the administration database for a particular application. The purpose of introducing these lists is to optimize and ease the process of comparing the new settings from the `settings.xml` file with the old settings already stored in the database. Also the lists serve the purpose of administrating which settings are stored in the database, and which are stored in the new `settings.xml` file. An example of how upgrade of `boolean` settings is handled, is seen in Listing 10.15.

```
1 public void handleBoolean(String varName, boolean varValue) {
2   BooleanSetting setting = booleanSettings.get(varName);
3
4   if (setting == null) {
5     DatabaseHelper.insertBoolean(db, appName, varName, varValue);
6   } else {
7     booleanSettings.remove(varName);
8   }
9 }
```

Listing 10.15. Shows an extract of `sw6.admin.parser.UpgradeSettings` used when upgrading boolean settings in administration database.

The reference `booleanSettings` seen in line two and seven in Listing 10.15, holds a reference to the list of all old settings of type `boolean` that was loaded by the `init(...)` method. The list is represented as a HashMap where the keys are the names of the `boolean` settings, and the values are instances of the `BooleanSetting` wrapper-class, which encapsulates information about a `boolean` setting. The purpose of line two and four is to decide if there exists an old boolean setting with the same name as the current new boolean setting, being handled by the `SettingsParser` class. If this is not the case, the new boolean setting is inserted into the administration database in line five. However, if a `boolean` setting with the same name already exists, the new setting is ignored according to the rule stating that we want to keep as many data unchanged as possible after an upgrade has completed. For booleans, both `true` and `false` are valid values, and no `min` or `max` constraints can be specified for this type to force us to update the setting. Therefore, if a new boolean

setting from the `settings.xml` file already exists in the administration database, no further action in the database layer is taken. Line seven removes the current boolean setting from the list of old boolean settings, if it occurs in the list.

By continuously removing old settings matching the name of a new setting, from the lists of old settings, the lists of old settings will at the end only contain old settings that are not declared in the new `settings.xml` file. The remaining settings in the lists of old settings are therefore considered to be unused settings that should be deleted from the administration database. In the `end()` method (see Figure 10.6), deletion of unused settings is carried out. This is done by traversing the lists of unused settings, and instruct the `DatabaseHelper` to delete each unused setting one by one.

The upgrade procedure of `boolean` settings is a simple case, and it only explains the optimistic rule saying that we want to keep as many settings unchanged as possible. What makes handling of `boolean` settings simple is, that it only allows two different values: `true` or `false`, and that it cannot be constrained by attributes specified in the `settings.xml` file. However, for settings like `integer` which also can be constrained by `min` and `max` attributes it becomes more complicated to handle the upgrade procedure.

The main rule when upgrading integers is, that if the old value is still valid according to the constraints specified for the new `integer` setting, then the old `integer` setting will be kept. However, if the old value does not comply with the constraints specified for the new `integer` setting, the setting will be reset to what has been specified in the new `settings.xml` file. In Listing 10.16 an extract of the code, handling upgrade of `integer` settings, is shown.

```
1  public void handleInteger(String varName, int min, int max, int varValue) {
2    IntegerSetting setting = integerSettings.get(varName);
3
4    if (setting == null) {
5      DatabaseHelper.insertInteger(db, appName, varName, varValue, min, max);
6    } else if (setting.min != min || setting.max != max) {
7      db.delete(DatabaseHelper.TBL_INTEGERS,
8        DatabaseHelper.COL_APP_NAME + " = ? AND " + DatabaseHelper.COL_VAR_NAME + " = ?",
9        new String[] { appName, varName });
10
11     if (setting.varValue >= min && setting.varValue <= max) {
12       DatabaseHelper.insertInteger(db, appName, varName, setting.varValue, min, max);
13     } else {
14       DatabaseHelper.insertInteger(db, appName, varName, varValue, min, max);
15     }
16   }
17
18   integerSettings.remove(varName);
19 }
```

Listing 10.16. Shows an extract of `sw6.admin.parser.UpgradeSettings` used when upgrading settings of type `integer` in administration database.

The principle of the code shown in Listing 10.16 is the same as for upgrade of `boolean` settings. Line two and four checks if the new setting is already present in the database, and inserts the new setting if that is not the case. The most interesting part of the code starts at line six, where it is checked if there is a difference between the old `min` and `max` constraints defined in the database and the new `min` and `max` constraints defined in the new `settings.xml` file. If the application developer has not made any changes in the boundary constraints for the new `integer` setting, the old value of the setting will be kept in the database.

However, if at least one of the `min` and `max` constraints has changed in the new version of the `integer` setting, then the integer setting is removed from the administration database in line seven. The setting is removed to ease the process of later inserting the `integer` setting again - this time with updated `min` and `max` constraints. The if-statement in line 11 checks if the old value of the `integer` setting complies with the new boundary constraints specified for the setting. If the old value of the setting complies with the new constraints, the old value of the `integer` setting is kept - this is seen in line 12, where the `integer` setting is inserted into the database again, with updated boundary constraints. However, if the old value of the `integer` setting does not comply with the new boundary constraints, the new default value of the `integer` setting is, along with

the new boundary constraints, inserted into the administration database. This happens in line 14. The new default value for the `integer` setting has been computed by the `SettingsParser` class, at the time where the `settings.xml` file was parsed. As this computation involves many rules and is very comprehensive, the computation of default values will not be described here. Instead we refer to Appendix B where a complete list of applied logic for the package handler is given. Here, it is also possible to find the rules stating the exact behavior of the `PackageHandler` when it comes to upgrade of settings.

10.4.6 Uninstallation

If a package is removed from the Android device, the `settings.xml` file is also removed, and all we have left, is the name of the removed package. Therefore, in order to remove the settings of an un-installed application, we cannot bring the `SettingsParser` into use. Instead, we have implemented a `deleteApp(...)` method in the `DatabaseHelper` which the `PackageHandler` invokes after a GIRAF package has been removed from the system. The method is seen in Listing 10.17.

```
1  public void deleteApp(String appName) {
2    SQLiteDatabase sqliteDatabase = this.getWritableDatabase();
3
4    sqliteDatabase.delete(TBL_BOOLEANS,   COL_APP_NAME + " = ?", new String[] {appName});
5    sqliteDatabase.delete(TBL_DOUBLES,    COL_APP_NAME + " = ?", new String[] {appName});
6    sqliteDatabase.delete(TBL_ENUM_ELEMS, COL_APP_NAME + " = ?", new String[] {appName});
7    sqliteDatabase.delete(TBL_INTEGERS,   COL_APP_NAME + " = ?", new String[] {appName});
8    sqliteDatabase.delete(TBL_OBJECTS,    COL_APP_NAME + " = ?", new String[] {appName});
9    sqliteDatabase.delete(TBL_STRINGS,    COL_APP_NAME + " = ?", new String[] {appName});
10   sqliteDatabase.close();
11   ...
12 }
```

Listing 10.17. Shows an extract of `sw6.admin.database.DatabaseHelper` used when all settings for a particular application is to be removed from the administration database.

As seen in Listing 10.17, the `deleteApp(...)` method declared in line one, makes sure that all rows having an application name equal to its `appName` parameter will be deleted from all the setting tables in the administration database. It should be noted that because the `enum` table uses foreign keys with cascades to reference elements in the `elements` table, the `enum` belonging to the application being removed, is automatically deleted.

11 Administration User Interface

The administration module provides a GUI, which can be used by a guardian to navigate and modify the values of administrative settings for specific applications, settings for the Android device, and properties related to the user of the GIRAF system. In this chapter we elaborate on how this user interface is generated and presented to the user on the Android device.

11.1 User Interface

Whenever the GUI of the administration module is launched, a list of all available GIRAF applications is displayed by the `MainActivity` placed in the `sw6.admin` package. Only applications having a `settings.xml` file with at least one visible setting are listed. A menu item pointing to the market place developed by group sw6c, as well as a menu item pointing to administrative settings, such as device settings and user profile, are also added to the list. When a user clicks on a menu item, e.g. an item representing an application, a new activity called `MenuActivity` is started, displaying the available settings and sub-menus of the application selected. If a sub-menu is pressed by the user, a new `MenuActivity` will be started displaying the available settings and sub-menus for the sub-menu selected. If a setting is pressed by the user, a dialog (or custom activity, depending on the setting) will be presented, giving the user an interface to edit the value of the setting. An example displaying a possible navigation scenario is presented in Figure 11.1, Figure 11.2, and Figure 11.3.

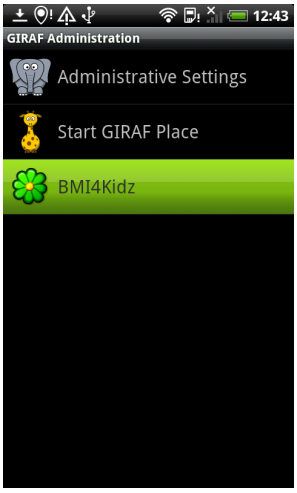


Figure 11.1. Root menu of the administration GUI. Here, administrative settings and GIRAF applications are listed. Pressing "BMI4Kidz " leads to Figure 11.2.

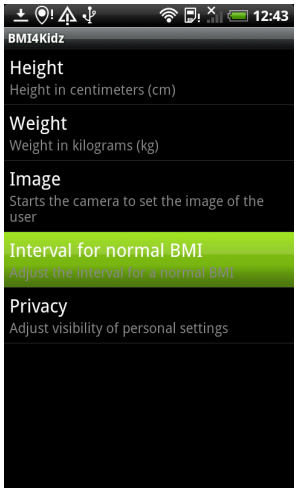


Figure 11.2. Here, the root menu for the settings declared in "BMI4Kidz " are listed. Clicking "Interval for normal BMI" leads to Figure 11.3.

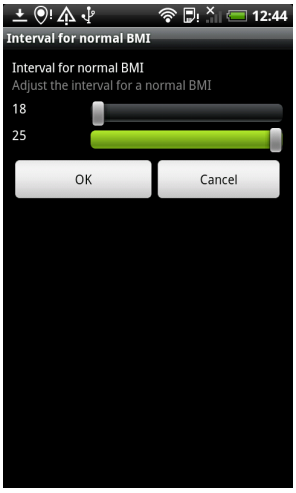


Figure 11.3. As "Interval for normal BMI" is a `stdobject` of type `sw6.lib.types.Interval`, the activity generating the GUI for adjusting the interval is loaded and shown to the user.

11.2 Auto Generation of User Interface

Because the settings for each application is unknown when the user interface is being programmed, a system for automatically generating the GUI for application settings needs to be created. Using the `settings.xml` file from a given application, the administration module can generate a GUI dynamically that can be used to browse and modify the value of settings. This GUI is generated by first parsing the `settings.xml` file using the `sw6.admin.gui.logic.SettingsParser` class. This results in a tree structure represented by an instance of the `Menu` class. In Figure 11.4, the `settings.xml` file for the BMI4Kidz application is represented as a `Menu` tree-structure.

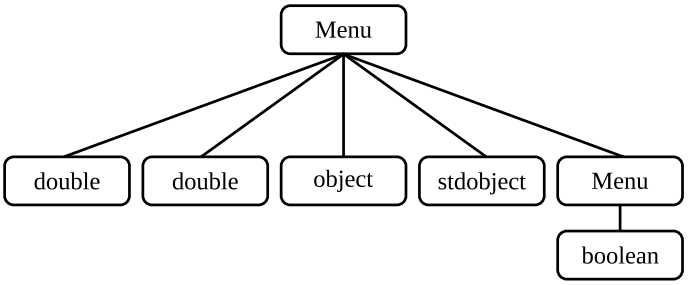


Figure 11.4. Menu structure for the BMI4Kidz application. A node is a menu and a leaf is a setting.

If the settings of the BMI4Kidz application (see Listing 8.2) are compared to the `Menu` structure seen in Figure 11.4, it can be seen that the tree-structure represents the settings and its nested menu. A leaf is an instance of the `Setting` class representing a setting parsed from `settings.xml`. `Setting` is a class that is used to encapsulate all the information of a specific setting, e.g. name, description, and minimum and maximum constraints. Using this tree-structure, the `MenuActivity` class generates the list of settings to be shown to the user, and creates an instance of `GuiBuilder` which is used to generate all necessary GUI dialogs so that they are ready to be shown. A GUI dialog enables the user to edit the value of a particular setting. Settings of type: **`double`**, **`integer`**, **`string`**, and **`enum`** will prompt the user with a dialog, if such a setting is to be edited. The `GuiBuilder` class is used by calling its `BuildDialogs` method. This method takes a `Menu` instance as a parameter. The source code of the `BuildDialogs` method is seen in Listing 11.1.

```
1  public void BuildDialogs(Menu menu, final Context context) {
2    ...
3    // Builds the dialogs for each setting
4    for (Setting setting : menu.getSettings()) {
5      // Creates an integer-double dialog
6      if (setting.isInteger() || setting.isDouble()) {
7        setupIntegerOrDoubleInputDialog(setting);
8      }
9      // Creates a string dialog
10     if (setting.isString()) {
11       setupStringInputDialog(setting);
12     }
13     // Creates an enum dialog
14     else if (setting.isEnum()) {
15       setupEnumDialog(setting);
16     }
17     ...
18   }
19 }
```

Listing 11.1. This code snippet shows the body of `BuildDialogs` where the settings of a menu is traversed to determine for which settings there should be created edit-dialogs.

The method in Listing 11.1 loops through each setting in the `Menu` object. It then builds a type-specific Android dialog that can be used to edit the value of the setting. As mentioned, there are four kinds of dialogs: integer -, double -, string -, and enum dialogs. Examples of how these dialogs look are given in the following.

The first dialog is for settings of type `integer`. The dialog is seen in Figure 11.5.

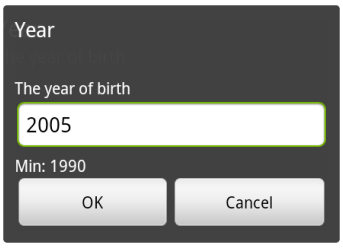


Figure 11.5. Dialog presented when the user wants to edit the value of a setting of type `integer`.

This dialog is presented to the user when editing the value of an integer setting. The value can be edited by clicking on the `EditText` field with a finger touch. An on-screen Android keyboard is then presented, allowing the user to edit. When the user has finished editing, the *OK* button is pressed to commit the change. If the user regrets the changes made the *Cancel* button can be pressed. Notice that below the `EditText` view, it says `min: 1990`. This is to inform the user that this setting has to be equal to or greater than 1990. If the user inputs an invalid value (e.g. 1988) the user will not be allowed to commit the change, as the *OK* button will disable automatically if a minimum and/or maximum constraint is not complied with.

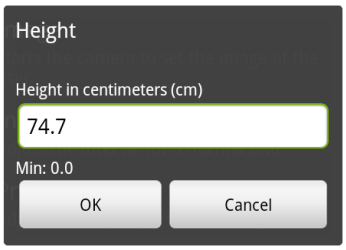


Figure 11.6. Dialog presented when the user wants to edit the value of a setting of type `double`.

The dialog seen in Figure 11.6 is presented to the user when editing a setting of type `double`. The same logic applies as for settings of type `integer`. This means that if the minimum and or maximum constraints are not complied with, the dialog will not allow the user to commit the change.

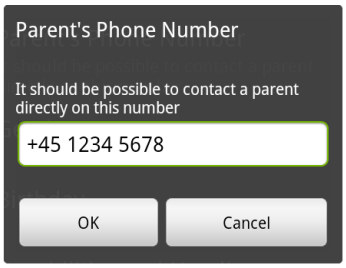


Figure 11.7. Dialog presented when the user wants to edit the value of a setting of type `string`.

When a setting of type `string` is to be edited, the dialog in Figure 11.7 is shown. For values of type `string`, any minimum and maximum constraints defines the minimum and maximum allowed length of the text string. Again, a keyboard will be shown on the screen if the user touches the text field.

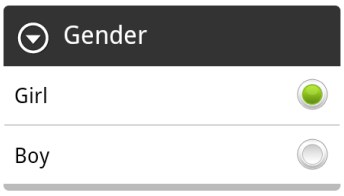


Figure 11.8. Dialog presented when the user wants to edit the value of a setting of type `enum`.

Finally, if a setting of type `enum` is to be edited, the dialog seen in Figure 11.8 is shown. In this dialog, the user can set a new value of the enum by selecting any of the available elements.

It should be noted that the colors of the Android GUI can differ, as this depends on the applied theme and visual layout of the device. In the GUI of the administration interface, this will affect the color of seek bars, selected items, the border of an input box and the like. However, it will not have a major impact on the design, as the layout will not be affected.

12 Test

In this chapter we describe how the library and the engine of the administration module have been tested. Different topics, techniques, and theories learned by attending lessons in this semester's project related course in TOV are used throughout the chapter. First, in Section 12.1, we describe different approaches to test and cover the related theory. Later in the same section, we argue why Test Driven Development (TDD) has not been our approach to testing, and instead we give a description of the alternative approach we have used. Then, in Section 12.2 and Section 12.3 we introduce regression- and unit testing respectively. We describe how and why unit tests have been used, and we give an example of one of the unit tests executed. Furthermore, we describe the idea of regression testing and how we used it to ensure that correcting bugs did not introduce other bugs in the system. In Section 12.4 we describe the use of integration testing in our project, and how we developed a tailored integration test, for testing the logic of the package handler. Finally, in Section 12.5, we describe how code coverage analysis can be performed on the Android platform, and how the results of the coverage reports have been used in our effort to get good coverage percents for our unit tests.

12.1 Approaches to Test

As covered in [Pat06, Chapter 4] the approach to software testing can be described by the following terms.

Black box Refers to testing a piece of software where the tester only knows what the software is supposed to do. The tester cannot look inside the box (the source code) and see how the code operates.

White box Refers to testing where the tester has access to the code being tested and is able to see how it operates. The tester can for example use this insight to test for certain cases, and to check which code has been covered by a test suite.

Static Refers to testing code that is not running.

Dynamic Refers to testing code that is running.

Combining these four terms gives the relationship seen in Table 12.1.

	Static	**Dynamic**
Black box	Specification review.	Testing the software against its specification.
White box	Code review, walkthroughs, and inspections.	Unit testing and code coverage measures.

Table 12.1. Overview of the relationship between the testing terms.

Static black box testing concerns testing of a specification. Here, a tester will check if the specification meets attributes like consistency, relevance, and accuracy. A bug in a specification could be an unclear

or weak requirement. Such bugs open for different interpretations by the programmer and might lead to functionality being implemented incorrectly. Finding problems in the software specification early on in the project is good, as it may prevent software bugs that are more expensive to fix.

Dynamic black box testing concerns testing of running software without knowing how the software is implemented. Typically this is done by entering input and validating the output against what is specified in the specification.

Static white box testing is about taking a closer look at some of the code that has been written. Code reviews could be handled by two programmers looking at each other's code. It could also be a code walkthrough where the developer who wrote the code formally presents the code line by line to a small group of other programmers and testers. Another approach could be doing code inspection. Here, another person than the original developer is going to present the code to a group of persons called the inspectors. Each inspector has the task to interpret the code from a different perspective, e.g. from a user, tester, or a product support perspective. Doing this, gives different views of the code under review and can be very helpful to spot different types of bugs. Furthermore, letting another person than the original developer present the code, forces the person to learn and understand the code, and doing this, might result in another interpretation that could help locating bugs [Pat06, Chapter 6, P 95].

Dynamic white box testing is about testing units of the software and measuring how much of the code has been tested by the written tests. In Section 12.3, unit testing will be described in greater detail and it will be covered how unit testing has been used in our project. Furthermore, in Section 12.5, some of the concepts of code coverage are described, and it is explained how code coverage has been used in our project.

12.1.1 Our Approach to Test

In the beginning of the development phase of our project we started to figure out how testing could be performed on the Android 2.2 platform. Here we found that dynamic white box testing using JUnit 3 was possible and we discussed whether to apply the approach of TDD or not. As the Android platform was new to all of us we concluded not to practice TDD as writing tests before the implementation might cause a lot of disturbance in trying to be confident with the new platform. However, we liked the idea of applying tests concurrently with the development of functionality in our project. Therefore, we decided to start programming and write unit tests as the development started to progress. In general unit tests were written for the most critical parts of our code in the library and the engine, including the database layer as well as the main interface of the library (`sw6.lib.Settings`). A couple of weeks into the development phase, we decided that before an iteration could be considered completed, all unit tests should pass.

As we became more confident with the new Android platform and its functionalities, we introduced other types of dynamic white box testing including automated integration testing as well as code coverage analysis. In the following sections it is covered how the different test approaches were covered in our project.

12.2 Regression Test

After a bug has been fixed or code has been modified in an implementation, it is good practice to do regression testing [Net11]. In regression testing, the already existing test cases are re-runned with the purpose of ensuring that recent changes did not have any unintended side-effects in code that is already working [Por]. In our project we used the defined unit tests covered in Section 12.3 and the defined integration test in Section 12.4 to carry out regression testing. Typically, regression testing was done after certain functionality had been implemented, or after a group of bugs had been fixed. Using regression testing helped us ensure that our changes did not affect any of the code that was already working as expected.

12.3 Unit Test

Linking untested modules of software together might be an error prone process. If errors occur during this process, it will be difficult to locate them as they may reside in the interface between the modules, or deep in one of the modules being linked. To avoid such messy situations, unit testing can be applied. Here, the code is built and tested in pieces (units) and as bugs are located and fixed, the units are gradually integrated into groups of modules on which integration testing is performed. If bugs are found during integration testing, the probability that these errors are caused by the interface of the linked modules is quite high, as the linked modules already have been tested in isolation. Surely there are exceptions to this rule, but in general, by applying unit testing a development team will have an easier job isolating bugs and have a greater chance of avoiding unpleasant situations when code units are linked together [Pat06, Chapter 7 P 109].

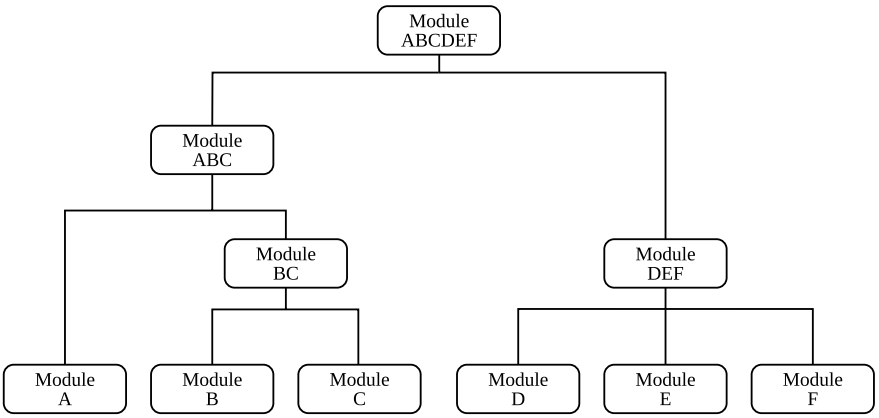


Figure 12.1. A system divided into individual pieces of code. Each piece is tested separately as a unit. Multiple pieces are then integrated and tested again.

The system seen in Figure 12.1 can be tested in different ways. An option will be the use of a bottom-up approach. Here subsystems are tested in isolation and integrated as we traverse up the tree. The pro of the bottom-up approach is, that it enables the development team to do early test of low level functionality [T11, Slide 25]. However, the cons of this approach are, that it defers testing of the complete system, and it requires the development team to keep redoing tests to ensure that the system is working as more and more units are grouped. The opposite approach is a top-down integration. Here the top modules are implemented first, and while working down the tree more and more subcomponents are to be implemented. Doing testing top-down demands some sort of emulation of the sub-components that are not yet implemented. Therefore, using a top-down approach might be an error prone process as writing emulations could be complicated; however if the emulated subcomponents are implemented correctly, a top-down approach always indicates if the system in overall is working as intended. This is different to the bottom-up approach, as the developers here are building up, and in the end, the final modules are integrated and the system as a whole is to be tested.

12.3.1 Our Approach to Unit Test

In our project we have utilized unit and integration testing on the Android platform using a bottom-up approach based on JUnit test suites as well as a tailored integration test described in Section 12.4. We have chosen the bottom-up approach as it was important for us, as library developers, to start at the lowest level and get some minimal functionality working such that other groups could get started giving input and accommodate to the library as it evolved. In the beginning of the project we worked with the library as well as the database layer of the administration module. To develop solid code, we wrote unit tests to ensure that the library and the database layer were working as intended. To avoid running buggy tests, we took our time to ensure that the tests were implemented correctly and was of high quality. Doing so helped us

locating bugs in the code under test, and because of the high quality of the tests, we knew that the bugs located, actually were bugs, and not false positives identified due to buggy tests. Furthermore, using unit tests allowed us to do regression testing, which made us able to check if new code had induced bugs in already working code. As time elapsed we got more and more modules implemented and we kept writing unit tests. In Listing 12.1 an example of one of our JUnit test cases is shown. In this test, we test if the database layer throws an `SQLiteException` if a setting that already exists in the database is being inserted again.

```
1  public void testInsertException() throws Exception {
2
3    // FLAG used to monitor if the first int has been inserted
4    boolean hasInsertedFirstInteger = false;
5
6    // Test for SQLiteException on insert for all types
7    try {
8      // At this point, the db is empty, so the setting is inserted here for the first time
9      dbh.insertInteger(appName, varName, 1337, 0, 2000);
10     hasInsertedFirstInteger = true;
11
12     // The setting is inserted here for the second time
13     dbh.insertInteger(appName, varName, 1337, 0, 2000);
14
15     // If we managed to get here, something went wrong
16     throw new Exception("Test failed.");
17
18     // We expect an SQLiteException to the thrown when the setting is inserted second time
19   } catch(SQLiteException sqle) {
20     if(hasInsertedFirstInteger == false) {
21       // The exception was thrown at first insert
22       throw new Exception("Test failed.");
23     } else {
24       // Everything went as expected
25       return;
26     }
27   }
28 }
```

Listing 12.1. Example of a JUnit test case testing if the database layer handles duplicated variables correctly. The test case is a part of the test suite in the `sw6.lib.test.DatabaseHelperTest` class.

The test is based on insertion of a setting of type `integer`. However, as the database layer is utilizing the same insert procedure for insertion of all types, this test is implicitly checking if the database layer in general is handling duplicated settings correctly. It should be noted that this test is not validating the primary key constraints of all types of variables in the database, as the case in Listing 12.1 is based on a certain type. To test the database and its unique constraints for all supported types, other tests like Listing 12.1 should be written and be based on the variable types to check. However, besides being trivial, these kinds of tests seems to be overkill to write in regards to how simple the logic is to verify in the DDL statements for the tables in question. Therefore it was decided not to write tests for all the primary key constraints in the database.

12.4 Integration Test

Linking units together into a group of modules as illustrated in Figure 12.1 demands testing to ensure that the modules interface correctly with each other. Testing groups of modules is referred to as integration testing [Pat06, Chapter 7]. This section will describe how integration testing has been used in our project. As we have worked with test design and test cases in the project related course in TOV, this section will be divided into a test design section (Section 12.4.1), a test case section (Section 12.4.2), and a test case execution results section (Section 12.4.3), to reflect a proper way of defining tests and report its results.

12.4.1 Test Design

This test design section covers how we designed the integration test for the package handler located in the administration module engine. The package handler is responsible for handling system-specific settings for packages being installed, upgraded, reset, or removed from the system. Details of how the package handler has been implemented and which functionalities it provides are described in Section 10.4.

12.4.1.1 Features Tested

It was to be tested if the package handler in the administration module engine handled settings for packages being installed, upgraded, or removed, in accordance with the logic specified in Appendix B.

12.4.1.2 Side Effects

When testing certain functionality this might result in other parts of the system being tested - this is known as side effects. Testing the features mentioned in Section 12.4.1.1 results in the following side effects.

Validation of settings.xml
As mentioned in Section 8.2, an application developer can specify settings for his application that he wants stored in our database. The settings can be of different types, e.g. integers and doubles and can be specified with optional min and max constraints as well as default values. The settings are specified in a `settings.xml` file and are validated against different rules to verify that it conforms to present definitions and grammars. In the process of installing or upgrading a package, the `settings.xml` file is validated against a DTD as well as against certain logic that we have specified must hold for this file - see Appendix B. Testing the validation process of `settings.xml` is a side effect of testing if the package handler can parse and store settings for packages being installed or upgraded.

Request, Insertion, and Update of Data through the Administration Module Library
To query or update settings stored in the database of the administration module engine, the package developer must use the administration module library. When the package developer is querying or updating settings through the library, it utilizes the content provider as well as the database layer of the administration module engine. As testing installation and upgrade of packages includes querying and updating of variables through the library, testing the library as well as the content provider and database layer of the administration module engine will be a side effect of testing installation and upgrade of packages.

12.4.1.3 Approach

The approach of testing the package handler was divided into two sessions. First, while the library as well as the database layer of the administration module engine was being developed, unit testing was used to perform testing at this lower level. As covered in Section 12.3, JUnit test suites were created, and using dynamic white box testing, a series of unit tests were used to confirm that these low level modules were working before the package handler was to be integration tested. In the second session, the package handler of the engine was to be tested. To ensure that the features listed in Section 12.4.1.1 could be tested, a test environment was defined where it should be possible to:

- Remove packages having a `sw6.*` prefix.
- Install a clean version of the administration module engine (contains content provider and database layer).
- Install a clean version of a test container that should be used to control the testing.

Furthermore, the environment should allow:

- Installation of packages to test if the package handler parses and stores settings correctly for packages that have just been installed.
- Upgrade of packages to test if the package handler parses and stores settings correctly for packages that have just been upgraded.
- Removal of packages to test if the package handler removes settings correctly for packages that have just been removed.

12.4.1.4 Pass and Fail Criteria

For the integration testing of the package handler to pass, insertion, upgrade, and removal of settings during installation, upgrade, and removal of packages must happen in accordance with the rules specified in Appendix B.

12.4.2 Test Cases

This section will describe how we designed an integration test project allowing us to test the features mentioned in Section 12.4.1.1 using a test environment meeting the requirements given in Section 12.4.1.3.

12.4.2.1 Approach

As described in the development guide for Android developers [And11h], monkeyrunner is a tool that provides an API for writing programs controlling an Android device and any applications running on it. A monkeyrunner script is used for testing purposes. It is written in Python and allows the developer to install and uninstall packages as well as sending keystrokes, capturing and comparing screenshots etc.

To ease the process of testing the package handler of the administration module engine, a monkeyrunner script was seen as a suitable solution as it allows automation of the process of installing, upgrading, and removing packages on an Android device without prompting the user with dialogs needing to be confirmed. However, using a monkeyrunner approach for handling packages does not reflect the final usage situation where a user is installing, upgrading, or removing applications through the package manager on an Android device. Taking that into account resulted in development of a test container. The test container is an Android application, and it enables us to execute the integration test regardless of whether it is automated by a monkeyrunner script running on a PC, or if it is running on an Android device with no PC and monkeyrunner attached.

To meet the test environment described in Section 12.4.1.3, the test setup seen in Figure 12.2 was created. It should be noted that Figure 12.2 shows the integration test where the monkeyrunner is attached. Running the test without the monkeyrunner, will make the test container be the place from where the test is started and controlled.

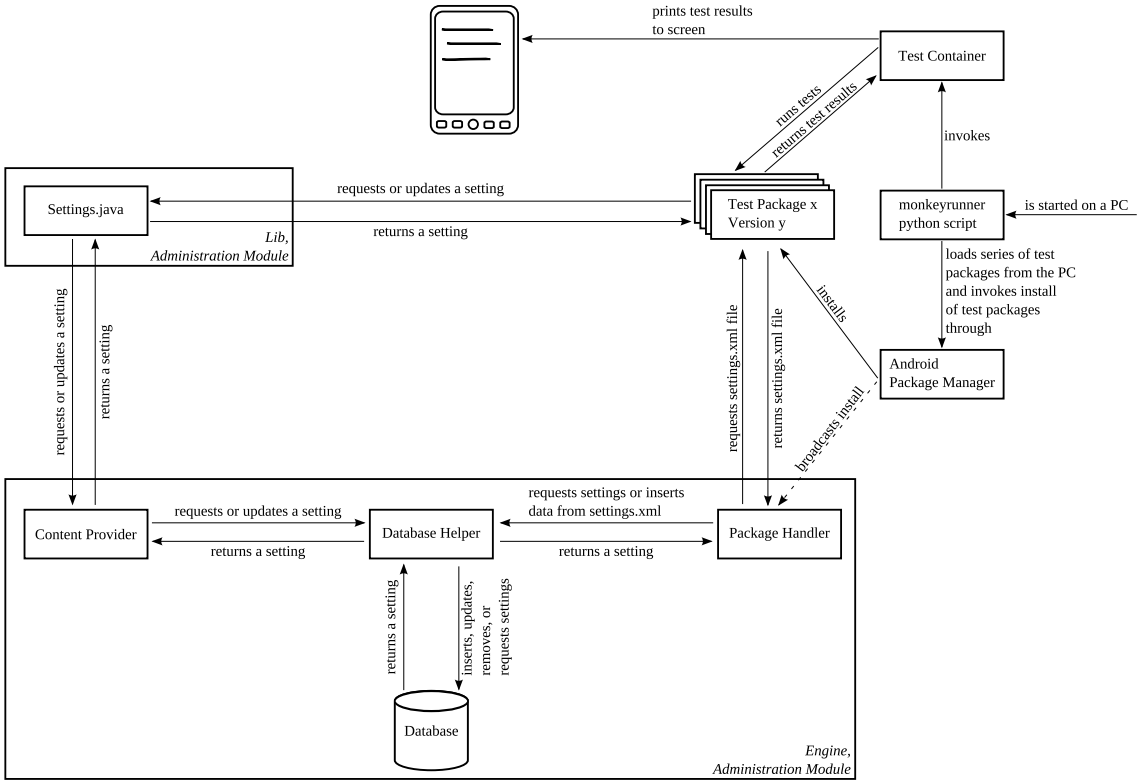


Figure 12.2. Integration test setup for testing the package handler in the administration module engine. The figure shows how installation and execution of tests is being handled by the monkeyrunner script, as well as by the library and the engine of the administration module. In the figure, it is assumed that the administration module engine has already been installed.

As indicated in Figure 12.2 the test starts in the monkeyrunner script. From here, all packages meeting the `sw6.*` prefix are removed from the Android device to ensure that the test is executed on a "clean" device. Then, the monkeyrunner invokes the Android Package Manager and instructs it to install the administration module engine followed by the first series of test packages. A test package is a small program containing a series of tests that verifies if the settings of the current test package have been handled by the package handler in accordance with the rules specified in Appendix B. In this case, where the monkeyrunner is attached, the test packages are located on the same PC as the monkeyrunner. Otherwise, in the case where the integration test is to be executed solely on the Android device using the test container, the test packages are loaded directly from the `/assets` folder of the test container. For the monkeyrunner to be able to invoke the tests, the test packages should be developed in accordance with the following criteria. A test package:

- Might have a `settings.xml` file defining settings to be stored in the database of the administration module engine. The `settings.xml` file might be empty or not present.

- Should have a single activity named "`TestActivity`".

  – The activity should be marked as exported to allow the test container (running in another process) to call the `TestActivity` in the test package.

  – The `TestActivity` must include the administration module library and use the `sw6.lib.Settings` class to check if the settings defined in `settings.xml` have been correctly handled by the package handler at the point where the test package has been installed or upgraded. The logic to test against is defined in Appendix B.

  - Finally, the intent-filter of `TestActivity` should include the following finite set of intent actions and categories:

    ```
    <action android:name="android.intent.action.MAIN"/>
    <category android:name="android.intent.category.DEFAULT"/>
    ```

    It should be noted that the `DEFAULT` category is used instead of `LAUNCHER`. This is done to prevent the test projects from showing up as standalone applications in the home menu of an Android device.

- Should return the test results collected by `TestActivity` to the test container which will output the results to the screen of the Android device running the test container.

After a test package has been installed by the Android Package Manager, the package manager sends a broadcast into the system informing that a package has been installed. The package handler of the administration module engine listens for broadcasts sent by the Android Package Manager which is indicated by the dotted line from the Android Package Manager to the package handler in Figure 12.2. When the package handler receives this broadcast, it asks the recent installed test package for its `settings.xml` file, and starts handling the settings in accordance with the rules listed in Appendix B. When this is done, the monkeyrunner script invokes the test container that is responsible for invoking the `TestActivity` of the test packages installed. The monkeyrunner instructs the test container to do so by clicking on a button in the test container, and the tests are now executed one by one. For each test package executing its `TestActivity`, it uses the administration module library to look in the database of the administration module engine to test if the settings of the current test package have been handled correctly.

Doing these tests underlines the importance of having unit tested the administration module library as well as the content provider and database layer of the engine before the integration test was carried out. With unit testing done for these modules, it is easier to debug and isolate errors during the integration test, as any errors found will have a high possibility of being related to the interface of the integrated modules. After a test has finished, the results are sent back to the test container, printing the results to the screen. For all available test packages, this procedure is done over and over again. It should be noted that the monkeyrunner first looks for test packages in version 1, for instance `sw6.app1v1` or `sw6.app2v1`. When all packages with version 1 have been installed and tested, the monkeyrunner proceeds to the next version, `sw6.app1v2` for instance, and starts to upgrade the packages installed. In this way, we are able to test upgrade as well as removal of packages, as long as the test specified in the test packages reflects how the settings for a particular test application should be represented in the database of the administration module engine. It should be noted that testing of removal of packages is simulated by upgrading to a test package containing no `settings.xml` file. Doing this, invokes the same procedure used when a package is upgraded, and as the new package does not contain any `settings.xml` file, the upgrade procedure will remove all settings for the particular application from the engine database. Using this simulation means that we do not test the exact code that handles removal of packages, but instead we test if the upgrade procedure is able to remove settings from the database. If removal of packages should have been tested without need for simulation, the monkeyrunner script and the test container should have been designed for this. The logic of the package handler's upgrade procedure is described in Section 10.4.5.

Based on this general description of the test procedure to perform, the detailed functionality of the monkeyrunner script can be defined to be:

1. Remove all `sw6.*` packages from the device.
2. Install the package containing the administration module engine.
3. Install the package handler test container application.
4. Install/upgrade a set of test applications having the same version, e.g. `sw6.app1v1`, `sw6.app2v1` etc.

    - As each test package includes the administration module library, the library source code will be copied into each test project and thereby be accessible by each test application.

5. Run the `TestActivity` for all `sw6.app[0-9]{1}v[0-9]{1}` packages installed on the device.
    - As the database creation is triggered when some application requests / updates data in a non-existing database for the first time, a database should be automatically created here if this is the first time that this step is executed.
6. Print the test results to the screen of the Android device.
7. While there exist newer test packages, go to item 4.
8. Empty the database of the administration module engine by removing all installed test packages.
    - This is done to create a new test scenario where there exists a database from the point of where applications are being installed for the first time.
9. Carry out step 4 to 7.
10. Done.

12.4.3 Test Results

In Table 12.2, the results of running the integration test for the package handler are shown. The results reflect the latest run of the integration test. From Table 12.2 it can be concluded that all test passed meaning that the package handler is working in accordance with the logic specified in Appendix B.

			Logic being tested													
	ID	**Test Package**	**§1**	**§1.1**	**§1.2**	**§1.3**	**§2**	**§2.1**	**§2.2**	**§2.2.1**	**§2.2.2**	**§2.2.3**	**§2.2.4**	**§2.3**	**§2.4**	**§3**	**§4**
Integration Test Results	**IT1**	sw6.app1v1	✓	✓	✓	✓											
	IT2	sw6.app1v2					✓	✓	✓		✓						
	IT3	sw6.app1v3					✓	✓	✓	✓					✓		
	IT4	sw6.app2v1	✓	✓	✓	✓											
	IT5	sw6.app2v2					✓	✓	✓	✓							
	IT6	sw6.app3v1															
	IT7	sw6.app3v2	✓	✓	✓	✓				✓	✓	✓	✓		✓		
	IT8	sw6.app4v1	✓	✓	✓	✓										✓	✓
	IT9	sw6.app5v1	✓	✓	✓	✓											
	IT10	sw6.app5v2												✓			

Table 12.2. Shows the results returned by each test package used to test the implementation of the package handler in the administration module engine. The symbol: ✓ indicates if the test of a certain logic passed. The symbol: ÷ indicates if the test for a certain logic failed.

12.5 Code Coverage

Code coverage is used to determine how well a set of tests covers the code being tested. Typically a code coverage analyzer tool is used when performing code coverage analysis. While running the tests, the tool runs in the background collecting statistics about which code were covered by the executed tests. Using these statistics can help a developer to:

- See which parts of a software solution are not being covered by the tests executed, and help writing new tests or spotting dead code.
- Check for redundant tests, testing the same partition of cases.

Also, code coverage helps a developer get a feeling of the overall quality of the software. If 90% of a solution is covered by the tests, and no bugs are found, the solution might be in good shape. On the contrary, if only

half of a solution is covered and still a lot of bugs are found, this might indicate that there is still a lot of work to do. However, even though a solution is 90% covered by tests, this might be a result of what is called the "Pesticide Paradox" described by the American software engineer and author Boris Beizer. The paradox says:

> *"Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual. "* [Bei90, Chapter 1, Section 7]

As explained in [Pat06, Chapter 3, P 41], the paradox describes the phenomenon that the more software is tested, the more immune it becomes to the tests that are run over and over again. The same thing happens to insects with pesticides. If the same pesticide is applied over and over, the insects will eventually build up resistance causing the pesticides to no longer work. The point in citing this paradox in relation to code coverage is that even though a solution gets a good coverage with a low amount of bugs, the developer should not be blinded by the statistics, as the reality might reveal that adding new tests with another equivalence partition might introduce a lot of new bugs.

There are different forms of code coverage. *Statement coverage* (or line coverage) concerns the amount of lines covered by a test. In Listing 12.2, an example of a method `isWeekend(...)` written in Java is seen. To reach 100% statement coverage of this code, a test could run `isWeekend("friday",true)`.

```
1 public boolean isWeekend(String day, boolean hasCommittedToTrunk) {
2   boolean isWeekend = false;
3   System.out.println("Checking if it is weekend...");
4   if(day.equalsIgnoreCase("friday") && hasCommittedToTrunk == true) {
5     isWeekend = true;
6   }
7   System.out.println("Check done.");
8   return isWeekend;
9 }
```

Listing 12.2. A dummy method `isWeekend` checking whether a development team has weekend or not.

The con of statement coverage is that it cannot tell if a test has covered all branches through the code being tested. In *branch coverage* this is resolved as it takes all available paths through the code into account. This means that repeating the test mentioned before will give a branch coverage of 50%, as the case where the **`if`** statement in line four evaluates to false is not tested. So to reach 100% branch coverage for the code in Listing 12.2 a test like `isWeekend("friday",false)` should be added to the test suite too.

As branch coverage only is concerned about the possible branches through the code, a more thorough type of code coverage is *condition coverage*. Here, all conditions where the code branches are taken into account. For Listing 12.2 this means that the test cases listed in Table 12.3 should be executed to reach 100% condition coverage.

Test No.	`day`	`hasCommittedToTrunk`	Conditions Covered	Line Numbers Covered
1	"monday"	false	false-false	1,2,3,4,6,7,8,9
2	"friday"	false	true-false	1,2,3,4,6,7,8,9
3	"monday"	true	false-true	1,2,3,4,6,7,8,9
4	"friday"	true	true-true	1,2,3,4,5,6,7,8,9

Table 12.3. List of conditions to check to reach 100% condition coverage in Listing 12.2.

As seen in Table 12.3, the condition coverage exercises the **`if`** statement of Listing 12.2 in line four by trying the four possible cases of its conditions.

12.5.1 Code Coverage Analysis for Android Applications Using EMMA

EMMA is an open source code coverage tool for Java, and supports the following code coverage types: class, method, basic block, and line. The main parts of this section will describe EMMA. Unless otherwise stated, the information will be based on the official project site: [Rou06].

To be considered for *class coverage*, a class must have been loaded and initialized by the underlying JVM. As a minimum, this implies that the class constructor is called. A class can be considered covered even though none of its other methods has been executed. Reporting class coverage can be used to spot dead code, or used to notice the developer that more tests are needed to cover untouched classes. In *method coverage* a method is considered covered when it has been entered. Looking at the number of uncovered methods can be a good indication of dead methods or missing tests.

Basic block coverage is the fundamental coverage of EMMA. All other types of coverage are derived from this. Basic block coverage concerns the percent of basic blocks covered in a class. In Java, a basic block is a sequence of bytecode instructions without any jumps or jump targets. As branching logic is added to a Java program, more basic blocks will be created. EMMA uses basic block coverage to generate more precise line coverage reports, as the number of covered blocks can help determine to which extent branches have been covered.

Line coverage is based on basic block coverage, and corresponds to branch coverage. Based on data from the basic block coverage run, EMMA takes into account the possible branches that a line can branch into, and computes a line coverage percentage based on the branches covered by the tests.

In Android version 2.0, an Ant build system with support for the EMMA project [And] was added. However, as no documentation has yet been provided by Google Android, and as the standard Ant builds generated for Android projects does not mention the option to enable code coverage, it is hard to believe that EMMA is officially supported. Therefore, to make EMMA work with Android a lot of unofficial blogs, guides, etc. had to be read and finally we managed to get EMMA working with Android. In Table 12.4 the results of running code coverage analysis for the unit tests, which we got for the administration module engine and library, can be seen. EMMA generates its code coverage reports as HTML files. The results seen in Table 12.4 is from a code coverage analysis executed April 22nd 2011. The latest code coverage report generated can be located on the enclosed DVD.

The first class `sw6.lib.types.Interval`, is a `stdobject` that we provide to the application developers, allowing them to represent an interval in their settings. `sw6.admin.database.DatabaseHelper` is a class containing all database logic, e.g. for creating the database, creating tables, inserting settings, etc. `sw6.lib.user.Profile` encapsulates all properties available for the user of GIRAF and retrieves them by lazy-loading. Finally, `sw6.admin.gui.logic.SettingsParser` is a class used to parse the `settings.xml` file, and return a `Menu` object representing a GUI to be auto generated by the administration module engine. Common for all four classes is that they are vital for the functionality of the administration module engine and library, which is why unit tests have been written to ensure working code.

	Coverage, %			
Name	**Class**	**Method**	**Block**	**Line**
`Interval`	100% (1/1)	100% (12/12)	100% (208/208)	100% (39/39)
`DatabaseHelper`	100% (1/1)	98% (43/44)	98% (878/921)	95% (183.2/193)
`Profile`	100% (1/1)	100% (17/17)	98% (127/129)	96% (25/26)
`SettingsParser`	100% (1/1)	60% (3/5)	93% (457/492)	90% (94/104)

Table 12.4. Shows the code coverage of unit tests executed for the classes listed. Executed April 22nd 2011.

As seen in the code coverage metrics in Table 12.4, the general basic block coverage is above 90%, and with all tests passing, this indicates that the code of the four classes is solid. However, as the pesticide paradox

stated, the classes might have become immune to the tests. To avoid this, we have continuously developed new tests with different equivalence partitions. This have ensured that new data has been used as basis for tests, and as the tests kept passing, this strongly indicates that we have some solid code working as intended.

Part III

Recapitulation

In this part we recapitulate on the entire multi project as well as on our development of the administration module. In Chapter 13 we evaluate the use of our customized development method. This will be done by taking the SOE course into account, which we have attended this semester. Following that, we describe future work of the administration module in Chapter 14, e.g. how to optimize the use of our XML validator as well as how the database layout of the administration engine can be improved. In Chapter 15, we conclude on our project and see if it meets the requirements specified in the system definition and if it meets the multi project architecture. Furthermore, in this chapter we also try to assess if the multi project in general meets its system definition. Finally, in Chapter 16 we reflects on the group collaboration during the multi project and suggests different improvements that can be taken into account in future multi project semesters.

Development Method

13

In this chapter we recapitulate on our custom development method used during the development of the administration module for GIRAF this semester. First, in Section 13.1 we describe our use of daily stand-up meetings and how they affected the development. Secondly, in Section 13.3, we give an overview of how testing has been applied, and which type of testing we took into use. Following that, Section 13.4 evaluates on the process of documenting the administration module. And finally, in Section 13.2, we evaluate the development process and describes the pros and cons of having one week iterations.

13.1 Daily Stand-up Meetings

In the development phase we made great use of stand-up meetings. At these meetings we were good at explaining the progress of each of our tasks, and explaining what actually had been done and what was to be done. Most of the time we did not ask all the questions that was originally intended, instead we "invented" our own questions and made sure that we got the necessary information covered during the meeting. For example, instead of asking "have you learned or decided anything new, of relevance to some of the team members?", we often brought this topic into the meeting if one of us actually had learned or added something new and interesting to the project. Furthermore, we did not attend "daily stand-up meetings" every day because we did not deem it necessary. If this was a real software project in a real software company, it would make sense to have the meetings every day. But since we are studying at a university where we do more than just develop software (e.g. attend courses or work on mini-projects), it was not necessary to have a meeting every day. Also, since we are a very small group, internal group communication was sometimes so strong that arranging daily meetings was not necessary.

13.2 Development Process

In the beginning we struggled with our plan of working with iterations from Friday to Friday. The problem was that our estimations were too optimistic. With the optimistic estimations we got a couple of Fridays where we had to work very late into the night in order to meet the scope of the current iteration. Having these late Fridays was for some of the group members very de-motivating and tiring. As we got more and more experience with the new Android platform, we got better at estimating the tasks to be done. This resulted in iterations with fewer tasks, and enabled us to complete iterations on schedule without any late Fridays. Using iterations with length of one week had pros and cons. The pros of using one week iterations were that the other groups in the multi project had the opportunity to review, use, and test our recent code and make suggestions for new features. The idea of this approach seemed very good on the paper, however we experienced that the other groups did not make that heavy use of our code and the documentation that we provided week by week. From this point of view it could have been beneficial if we had used iterations of at least two weeks. In this way we could have minimized the overhead of using time on committing trunk releases, updating documentations and guides, and instead been focusing on writing code and solve more tasks.

13.3 Testing

Testing has been carried out in the project with great success. Besides using JUnit for testing the administration module, we also used unit tests to cover testing of the administration module engine. Every time after a bug has been fixed or code has been modified, we did regression testing to ensure that the fixes/enhancements did not introduce new issues into the system. As our project is highly module based, we also used integration tests to ensure that our units were properly integrated. Integration testing was carried out using technologies specific to Android - especially, a tool named "monkeyrunner" was taken into use. A python monkeyrunner script was written by us, and used for testing the logic applied at installation, upgrade, and removal of GIRAF applications. Furthermore, we also introduced evaluation of our tests by using code coverage analysis on the Android platform. Using the code coverage tool allowed us to analyze the coverage reports and get an idea of how good coverage our unit tests had. In the beginning, the code coverage reports were used to optimize a couple of unit tests, but as time elapsed, the code coverage reports showed that the important parts of our code were covered well, so only minor changes to our tests were made from that point.

13.4 Documentation

Continuously throughout the project we wrote documentation constantly. The administration module library was always well-documented in Javadoc, and through the wiki system at Google Code we always ensured that new documentations were available reflecting the latest functionality. However, even though we worked hard at writing thorough and good documentation that was continuously published at our wiki system, the other groups of the multi project neglected to read and use it. In one of our documentation documents we inserted an "Easter egg", saying that if "you are reading this, you have won a free piece of chocolate. Go to our group room to claim you price!" After more than one week, only a single person had spotted this - and this strongly indicates that documentation rarely is read. In Chapter 16 we reflect on the collaboration between the multi project groups, and here, this problem is further discussed.

Future Work

14

In this chapter we describe plans for improving the administration module of GIRAF. The reason why these plans were not implemented during the development phase is because we decided to focus our time and resources on other parts of the project. The goal of this chapter is to define the areas of the administration module that can be improved, and hopefully inspire future multi project semesters to take our advices into account and make the administration module even better.

14.1 Administration Interface for a PC

At the start of this project we planned to develop a PC program that would allow guardians to configure GIRAF devices. However, this idea was removed from our project scope midways because we decided to focus our time and resources on other more important parts in the administration module. The idea of the administration PC interface was that the PC program could be used to connect to the GIRAF device via Wireless Local Area Network (WLAN), allowing the PC program to configure the user profile properties as well as the administrative settings of GIRAF applications installed on the device. It was also intended, that the PC program should be able to store a copy of the administrative settings on the PC hard drive, enabling guardians to edit the settings, when the GIRAF device was not connected. And then, at the time when the GIRAF device is again connected to the system, the setting changes made offline should then be synchronized between GIRAF on the Android device and the PC interface on the PC.

Some parts of the PC administration system were developed before the idea was taken out of our scope. A prototype for the PC GUI was written as a Java program. A screenshot from the program is seen in Figure 14.1.

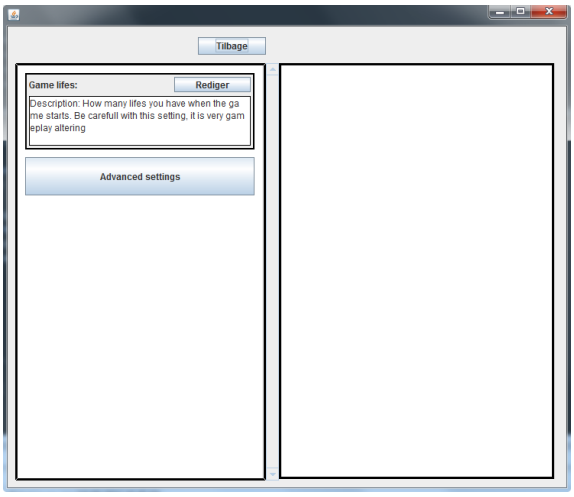


Figure 14.1. This is a screenshot of the Java PC GUI for remotely administrating the administrative settings in GIRAF.

The developed prototype could parse a `settings.xml` file, and display a GUI enabling the user to edit the value of these settings (much like the GUI on the GIRAF device). Furthermore, a network infrastructure was written, allowing the Android device to communicate with the PC program over WLAN.

14.2 Support for Default Values in Settings of Type Enum

Currently, if a GIRAF application developer defines a setting of type `enum` in the `settings.xml` file, the only way a default value for this setting can be specified, is by placing the element as the first in the list of elements inside the enum-setting. In some instances, the developer might want to choose a different default value for his `enum` setting while preserving a specific order of the enum elements. As this is not currently supported, this feature could be added in the future by allowing a "default" tag when declaring a setting of type `enum` in the `settings.xml` file. An example of this can be seen in Listing 14.1.

```
1 ...
2 <enum varName="gender" realName="Gender">
3   <element realName="Boy" value="1" />
4   <element realName="Girl" value="2" />
5   <default>2</default>
6 </enum>
7 ...
```

Listing 14.1. This is an example of how an enum setting might be defined with a default value.

In this case, the XML code in Listing 14.1 would give the setting `gender`, the default value of "Girl" as this enum element has the value of "2". Of course, in order for this to work properly, the `settings.xml` validator would have to support this feature too.

14.3 Different Ordering of Settings in the GUI of the Administration Module

When the administration GUI displays a set of settings, the settings are currently sorted according to setting type. Integers are displayed first, then strings, then doubles, etc. A better approach, would have been to list the settings in the order they have been specified in the `settings.xml` file, as this would allow the application developer to decide the order of settings shown in the GUI of the administration module. The problem in the current implementation lies in the `settings.xml` parser that the GUI uses. The parser starts by parsing all the integers, then the strings, then the doubles, etc. of the `settings.xml` file. By not taking the settings' position into account while parsing, and by sorting the settings by type in the GUI of the administration module, this means that the order of settings specified in `settings.xml` file are rarely met. This also means that GIRAF developers have no way of defining the exact order settings should be displayed in. This problem could be addressed by logging the order of when settings appear in the `settings.xml` file, and then order the settings in the GUI accordingly.

14.4 Reduce Data Redundancy in the Database of the Engine

Currently, all relational schema for settings in the database contains a column called `app_name`. The column is used to store the name of the application which a particular setting belongs to (the current structure of the database can be seen in Section 10.2). However, since an application can have multiple settings, the name of the application can appear in multiple rows. For instance, if an application has a setting of type `boolean` and a setting of type `string`, both the `booleans` table and `strings` table will now both contain the name of the application. Typically, data redundancy leads to data anomalies and corruption and generally

should be avoided by design [RC09]. A solution to this problem could be creating an extra relational schema called `applications`, like the one defined in Listing 14.2, and use a one-to-many relationship to represent the application names.

```
1 CREATE TABLE applications (
2     app_id      INTEGER PRIMARY KEY AUTOINCREMENT,
3     app_name    TEXT    UNIQUE
4 );
```

Listing 14.2. DDL statement for creation of a new `applications` table representing unique id's of each GIRAF application.

This means that only a single and unique `app_id` should exist for each application, and that this id should be used to identify a particular application in all tables in the database. In this way we can avoid data redundancy and get a database that is easier to maintain while ensuring its integrity is not violated. In Listing 14.3 an example of a DDL statement for a `doubles` table using the new `applications` schema is seen. Now, instead of storing the `app_name` in each row in the database, the rows will just contain a foreign key, pointing at a row in the `applications` table. This saves space and removes the problem of data redundancy. In line 10, the foreign key constraint to the application id in the `applications` table is declared.

```
1  CREATE TABLE doubles (
2      app_id    INTEGER,
3      var_name  TEXT,
4      var_value REAL,
5      min       REAL,
6      max       REAL,
7      timestamp BIGINT,
8      dirty     BOOLEAN,
9      CONSTRAINT 'doubles_primary_key' PRIMARY KEY ( app_id, var_name ),
10     CONSTRAINT 'app_id_fk_constraint' FOREIGN KEY ( app_id )
11             REFERENCES applications ( app_id )
12             ON DELETE CASCADE
13             ON UPDATE CASCADE,
14     CONSTRAINT 'min_max_constraint_check' CHECK ( min <= max
15             AND
16             var_value >= min
17             AND
18             var_value <= max )
19 );
```

Listing 14.3. DDL statement for creation of a new `doubles` table using a unique application id referenced in the `applications` table.

A nice feature that should be noted in Listing 14.3 is the use of cascades in line 12 and 13. By applying those, deleting settings becomes easier, as only the parent key in the `applications` table must be deleted to delete all settings of type `double` that referenced the key. And furthermore, if the id of the application should be updated in the `applications` table, this will also be reflected in the `doubles` table. Another neat thing about the use of the `applications` table in Listing 14.2 is, that it allows for adding properties for a specific application, e.g. a timestamp showing when the application has last been used etc.

14.5 Load Time for GUI in Administration Module

On older Android devices made for Android 2.1, and on some few devices running Android 2.2 natively, the GUI of the administration module loads slowly. The reason for this poor performance is found in the great amount of tasks that are executed when loading the root menu of the administration module GUI. Here, the list of all GIRAF applications are traversed and it is determined whether the application contains a `settings.xml` file or not. Furthermore, as the GUI only lists GIRAF applications with visible settings declared in the `settings.xml` file, the GUI is also responsible for parsing through each `settings.xml` file, and count the number of visible settings. If the count is greater than zero, the application can be shown,

otherwise, it will not be shown. The process of counting the number of visible settings utilizes a parsing method originally build and currently used for GUI generation based on a single `settings.xml` file - for instance, when a single application is selected, the GUI for this application's visible settings is generated. The parser builds an object structure reflecting the menu structure in the selected application's `settings.xml` file. By using the same parsing method to determine the number of visible settings, a large overhead occurs, as there is no need for a representation of the menu structure at this point. Therefore, to solve this issue, different solutions could be considered.

For instance, if the new database structure covered in Section 14.4 will be applied, the `applications` table could be used to indicate if an application contains visible settings. Of course, this would require the installation procedure to take care of this when inserting the application settings, however, with regards to the GUI this should give a much better performance, as there no longer will be need to repeatedly parse trough the entire `settings.xml` file to determine if visible settings exists or not. Another approach could be to re-think the idea of parsing the `settings.xml` file at runtime. An idea could have been representing the `settings.xml` file directly in the database, and only parse it once, and that should be during application installation. In this way there will be no need for a lot of XML parsing at runtime, which we think might improve the performance radically.

14.6 Generation of Java Classes from the settings.xml File

In the current version of the administration module, developers are supposed to use the library, which has been designed to communicate with the administration module engine. The library has no knowledge of the `settings.xml` file that comes with each application and thus no knowledge of the available settings. Therefore, the developers will have to include the variable name of a specific setting as a parameter, when trying to update or retrieve a setting through the library. During this process, the developer might misspell the variable name which could result in an error. To remedy this problem, the developer could declare a set of global constant variables to hold the variable names. However, this still opens up for the possibility of misspelling while defining the global constants.

The most error proof solution to the problem would be for us to extend the functionality of the `settings.xml` validation tool, so that after validating a `settings.xml` file, it would generate a Java class that could be included in the individual application. This class would then contain a get and set method for each setting in the `settings.xml` file, which could be used to update and retrieve the value of settings. In this way, variable names for settings would only have to be defined once in the `settings.xml` file.

Conclusion

15

In this chapter we first conclude on our project. The conclusion is based on the requirements for the administration module stated in the system definition of GIRAF. Then we give a brief conclusion on the multi project, based on the results of the multi project test.

From the system definition in Section 2.2 the following is stated about the administration module of GIRAF:

> *"... Guardians should be able to administrate the system by controlling selected application-, system- and user-specific settings through an administration interface on the phone. Based on these settings, as well as the location of the unit, a home menu should be responsible for providing access to applications that conforms to the current settings and the state of the system. It should be possible for any third party to develop and provide additional applications to the system..."*

From our point of view, the system definition states that the administration module should:

- Handle specific selected application settings.
- Handle system settings of the Android device.
- Handle user-specific settings.
- Provide an administration GUI on the phone, allowing guardians to edit the settings.
- Provide an interface to the settings such that applications can get and set their administrative settings.
- Handle an applications administrative settings when it is installed, removed, or upgraded on the Android device.

As covered in Section 8.2, declaring administrative settings are done using an XML file. This enables an application to declare which settings should be handled by the administration module, and only be accessible by the guardians. To enable a guardian to administrate the system- and user-specific settings, these options have also been added to the GUI of the administration module as described in Chapter 11. To allow applications to access the administrative system- and user-specific settings a library was developed for this particular purpose (see Chapter 9). Finally, handling administrative settings for new packages, upgraded packages, and removed packages, the package handler described in Section 10.4 was developed. Based on this list of requirements and our solutions, we can conclude that the administration module meets all the requirements stated in the system definition.

With regards to the multi project, we can only conclude by taking the multi project test into account. As it was described in Chapter 4, the conclusion of the test was, that based on the amount of errors found and their level of importance, it seemed reasonable to assume, that the GIRAF multi project, would be able to meet most of its requirements specified in its system definition.

Multi Project Reflection 16

Because an important part of the multi project is the cooperation with other groups and integration with other groups' solutions, we find it relevant to reflect on this. In this chapter we briefly describe how we experienced the interaction between the groups during this semester's multi project. The purpose is to reflect on what went good (Section 16.1) and what went wrong (Section 16.2) and describe what might be done differently in other multi projects.

16.1 What Went Good

One of the things that worked really well in this semester's multi project has been the use of Google Code and its supplied issue tracker. Especially the latter has been used extensively in the last part of the project where several of the groups were checking for flaws and other issues in each other's applications. We have even received bug reports from other groups where most of them have been corrected. Furthermore, we have also been able to use the issue tracker to make other groups aware of faults and deficiencies in their code. As a result of having code from different groups shared at one place, there were also issues where we had the energy to check where in another group's code the issues were introduced and enclose that information in our bug reports.

With regards to bug-tracking there has been a good cooperation between some of the groups which sometimes helped the groups to solve bugs together. If a multi project is to be carried out in the future, we are convinced that it will be a great help if the project includes a project management tool like Google Code, providing a common code base as well as a common issue tracker. In this way, the groups will be able to easier help each other, monitor project progress, and track the status of reported issues.

16.2 What Went Wrong

Besides the good experience with the use of the issue tracker, different other things went wrong in the collaboration between the groups during this semester. In this section, we account for some of the problems and try to suggest improvements to be made in future multi projects.

The first problem considers a single group, which decided to isolate themselves from the multi project community almost from the beginning of the semester. In particular, this involved minimal communication with the other groups as well as minimal use of the common project management tool at Google Code. Besides minimal contribution to the community of the multi project, a consequence of their isolated behavior was seen at the time when integration testing of the entire multi project was to be carried out. After having struggled with compatibility issues caused by the groups own policies and ways of doing things, the group had to give up being a part of the integration test, as their application could not integrate with the GIRAF system at the particular moment. It is impossible to determine whether common code-sharing would have solved a problem like this, but as developers of the administrative module, it is not easy to completely ensure

that everything works and functions as intended, when only one of the two groups developing applications contributes to the daily flow of the multi project by committing code to the shared code base at Google Code.

The other problem during the development in the multi project can be divided into many small problems all considering lack of interest in collaboration between the groups when bug tracking was not on the agenda. During our development of functionality in the library and engine of the administration module we weekly published documentation of all functionality as well as our plans for the next iteration. In this way, we more or less invited the other groups to affect our development, as we never hid what was actually being worked on and what was planned to be worked on in the future. In an attempt to make the other groups a more integrated part of our development, we continuously sent emails, where we asked the groups to read some of our material and provide comments to our plans for new functionality, as well as for functionality already implemented. In this way, the idea was to "invite" our fellow students to affect our module of the multi project.

However, the material we sent remained unread (only a couple of students took time to read it) and we tried to adjust our deadlines for when it should have been read and commented. However, this did not have any effect - the amount of feedback on our work was minimal. Therefore, half-way through the project, we decided to set a hard and final deadline for submission of new proposals for functionality to the administration module. After exceeding the deadline, no feedback was received. Therefore, by request from our supervisor, we arranged individual group meetings with each group and postponed the deadline for one week. In the invitation it was made clear that certain documentation should be studied before the meeting. Only at one of these meetings, the students had prepared and studied the documentation, and only one group had a feature request.

At the end of the semester people started to use our library and issues were reported. However, most of the issues were caused by the application developers who still have not read our documentation. From this it can be concluded that using a large amount of time writing documentation with the goal of helping the multi project and the fellow students, is more or less waste of time, as we continuously could conclude, that the majority did not look at it. We have no ideas of how this lack of interest in the multi project can be resolved.

The third and final problem concerns agreements and quality of code. The problem was that at the majority of the group leader meetings a single group promised too much functionality than they were capable of implementing. The result was low-quality code where the amount of errors constituted a major problem before the multi project integration test was to be carried out. This got the consequence that the integration test was postponed multiple times until the worst of the bugs were fixed. Even though the group did share their code using the Google Code code base, their promises and serious attitude on the surface made us postpone a look into their code. Furthermore, some of the implementations made by this group did not meet agreements made during the multi project. We think that an experience like this indicates the need for someone continuously monitoring the multi project, making sure that everything progresses as planned. A solution might be to introduce a supervisor for the entire multi project.

Bibliography

[And] Google Android. Android 2.0, Release 1 - EMMA. http://developer.android.com/sdk/android-2.0.html. Downloaded from site: April 23rd 2011.

[And08] Google Android. Announcing the Android 1.0 SDK, release 1. http://android-developers.blogspot.com/2008/09/announcing-android-10-sdk-release-1.html, September 2008. Downloaded from site: April 27th 2011.

[And11a] Google Android. Adding SDK Components. http://developer.android.com/sdk/adding-components.html, May 2011. Downloaded from site: May 4th 2011.

[And11b] Google Android. ADT Plugin for Eclipse. http://developer.android.com/sdk/eclipse-adt.html, May 2011. Downloaded from site: May 4th 2011.

[And11c] Google Android. Building and Running. http://developer.android.com/guide/developing/building/index.html, May 2011. Downloaded from site: May 4th 2011.

[And11d] Google Android. Code Style Guidelines for Contributors. http://source.android.com/source/code-style.html, May 2011. Downloaded from site: May 4th 2011.

[And11e] Google Android. ContactsContract.PhoneLookup. http://developer.android.com/reference/android/provider/ContactsContract.PhoneLookup.html, May 2011. Downloaded from site: May 11th 2011.

[And11f] Google Android. Content Providers. http://developer.android.com/guide/topics/providers/content-providers.html, May 2011. Downloaded from site: May 18th 2011.

[And11g] Google Android. Managing Virtual Devices. http://developer.android.com/guide/developing/devices/index.html, April 2011. Downloaded from site: May 4th 2011.

[And11h] Google Android. monkeyrunner. http://developer.android.com/guide/developing/tools/monkeyrunner_concepts.html, April 2011. Downloaded from site: April 22nd 2011.

[And11i] Google Android. Platform Versions. http://developer.android.com/resources/dashboard/platform-versions.html, April 2011. Downloaded from site: April 25th 2011.

[And11j] Google Android. SQLiteOpenHelper. http://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html, May 2011. Downloaded from site: May 14th 2011.

[And11k] Google Android. The AndroidManifest.xml File. http://developer.android.com/guide/topics/manifest/manifest-intro.html, May 2011. Downloaded from site: May 4th 2011.

[And11l] Google Android. What is Android? http://developer.android.com/guide/basics/what-is-android.html, May 2011. Downloaded from site: May 5th 2011.

[Bei90] Boris Beizer. *Software testing techniques*. Van Nostrand Reinhold Co., New York, NY, USA, 2nd edition, 1990. ISBN: 0-442-20672-0.

[Bei07] Lynn Beighley. *Head First SQL*. O'Reilly Media, Inc., 1st edition, 2007. ISBN: 978-0-596-52684-9.

[Car11] Mark Carter. Version of SQLite used in Android? http://stackoverflow.com/questions/2421189/version-of-sqlite-used-in-android, April 2011. Downloaded from site: May 5th 2011.

[DI09] Natur-og Sundhedsvidenskabelige Fakulteter De Ingeniør. Bacheloruddannelsen i Software. http://www.sict.aau.dk/digitalAssets/3/3331_softwbach_sept2009.pdf, September 2009. Downloaded from site: May 24th 2011.

[Fow04] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern. http://martinfowler.com/articles/injection.html, January 2004. Downloaded from site: May 15th 2011.

[Fre11] Derek Freeman. Rooting for Dummies: A Beginner's Guide to Rooting your Android Device. http://www.androidauthority.com/rooting-for-dummies-a-beginners-guide-to-root-your-android-phone-or-tablet-10915/, March 2011. Downloaded from site: May 4th 2011.

[JB11] Canada Janine Bernat, University of Regina. Crow's Foot Notation. http://www2.cs.uregina.ca/~bernatja/crowsfoot.html, May 2011. Downloaded from site: May 14th 2011.

[MMMNS01] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. *Object-oriented analysis & design*. Marko Publishing, 1st edition, 2001. ISBN: 87-7751-150-6.

[Net11] Microsoft Developer Network. Regression Testing. http://msdn.microsoft.com/en-us/library/aa292167%28v=vs.71%29.aspx, April 2011. Downloaded from site: April 22nd 2011.

[Pat06] Ron Patton. *Software Testing*. Sams Pulishing, 2nd edition, 2006. ISBN: 978-0-672-32798-8.

[Por] Adam Porter. Testing. http://www.cs.umd.edu/~aporter/html/currTesting.html. Downloaded from site: April 22nd 2011.

[RC09] Peter Rob and Carlos Coronel. *Database systems: design, implementation, and management*. Cengage Learning, 1st edition, 2009. ISBN: 978-1-423-90201-0.

[Rou06] Vlad Roubtsov. EMMA. http://emma.sourceforge.net/, January 2006. Downloaded from site: April 23rd 2011.

[SQL11a] SQLite. About SQLite. http://www.sqlite.org/about.html, May 2011. Downloaded from site: May 14th 2011.

[SQL11b] SQLite. Appropriate Uses For SQLite. http://www.sqlite.org/whentouse.html, May 2011. Downloaded from site: May 14th 2011.

[SQL11c] SQLite. Release History. http://www.sqlite.org/changes.html, April 2011. Downloaded from site: May 5th 2011.

[SQL11d] SQLite. SQLite Foreign Key Support. http://www.sqlite.org/foreignkeys.html, May 2011. Downloaded from site: May 14th 2011.

[SQL11e] SQLite. SQLite is Transactional. http://www.sqlite.org/transactional.html, May 2011. Downloaded from site: May 14th 2011.

[T11] Ulrik Nyman Test and Verification 3. Examining the Code and Testing the Software with X-Ray Glasses - Bottom-up testing, slide 25. https://intranet.cs.aau.dk/fileadmin/user_upload/Education/Courses/2011/TOV/test3ver2.pdf, March 2011. Downloaded from site: April 22nd 2011.

[TK04] Amrit Tiwana and Mark Keil. The one-minute risk assessment tool. http://www.cs.aau.dk/~jeremy/SOE2011/resources/TiwanaKeil.pdf, 2004. Downloaded from site: May 4th 2011.

[vH11] Dimitri van Heesch. Doxygen. http://www.stack.nl/~dimitri/doxygen/index.html, March 2011. Downloaded from site: May 10th 2011.

[W3S11] W3Schools. Introduction to DTD. http://www.w3schools.com/DTD/dtd_intro.asp, May 2011. Downloaded from site: May 10th 2011.

Part IV

Appendix

Acronyms

A

ACID Atomic, Consistent, Isolated, and Durable

ADT Android Developer Tool

API Application Programming Interface

APK Android Package

AVD Android Virtual Device

BMI Body Mass Index

DDL Data Definition Language

DNS Domain Name System

DTD Document Type Definition

DVM Dalvik Virtual Machine

GUI Graphical User Interface

HTML HyperText Markup Language

IDE Integrated Development Environment

IP Internet Protocol

JVM Java Virtual Machine

MPL Multi-Project Management (Multi-Projekt Ledelse (MPL))

OS Operating System

PECS Picture Exchange Communication System

ROM Read-only memory

SD Secure Digital

SDK Software Development Kit

SMS Short Message Service

SOE Software Engineering

SQL Structured Query Language

TDD Test Driven Development

TOV Test and Verification (Test og Verifikation (TOV))

URI Uniform Resource Identifier

URL Uniform Resource Locator

USB Universal Serial Bus

WLAN Wireless Local Area Network

WYSIWYG What You See Is What You Get

XP Extreme Programming

Applied Logic for Package Handler

B

B.1 Applied Logic for Handling Settings at Package Installation

Insertion of settings into the database should be carried out according to the following rules:

§1 When a package has been installed, its `settings.xml` file is parsed (if one exists), and its data is inserted into the database.

§1.1 If a database does not exist in the administration module engine at this particular moment, the settings will be inserted by a "first run procedure" when someone wants to get or set settings in the database for the first time.

§1.2 If a database do exists, the settings are inserted directly.

§1.3 If no default values are specified for settings allowing this, default values should be inserted in accordance with **§2.2.1** through **§2.2.4**.

B.2 Applied Logic for Handling Settings at Package Upgrade

Upgrade of settings in the database should be carried out according to the following rules:

§2 If a setting is already stored in the database and if it complies with the constraints specified in the new `settings.xml` file, then the database value of the current setting will be kept.

§2.1 If a setting already stored in the database does not comply with the constraints specified in the new `settings.xml` file, the setting will be reset according to the constraint and default value definition in `settings.xml`. However, to this, there follow some specific rules. Se **§2.2** and **§2.2.1** through **§2.2.4**.

§2.2 If a setting already stored in the database where min and max boundaries are supported, e.g. an integer, does not comply with the min and max constraints specified in a new `settings.xml` file, and if there is not defined a new default value in the new `settings.xml` file, the system should choose a default value according to **§2.2.1** through **§2.2.4**.

§2.2.1 While taking the optional min and max boundaries into account, a default value for variables of type integer is set such that it is as close to 0 as possible.

§2.2.2 While taking the optional min and max boundaries into account, a default value for variables of type double is set such that it is as close to 0.0 as possible.

§2.2.3 A default value for variables of type boolean is set to false.

§2.2.4 While taking the optional min and max boundaries into account, a default value for variables of type string is set such that it contains the minimum amount of letter spaces.

§2.3 Settings stored in the database that do not appear in the upgraded `settings.xml` file will be deleted from the database.

§2.4 Settings that do appear in the upgraded `settings.xml` but not in the database will be inserted into the database having the default values specified in the `settings.xml` file or a default value computed in accordance to **§2.2.1** through **§2.2.4**.

B.3 Applied Logic for Get and Set of Administrative Settings

Get and set of settings using the administration module library should be carried out according to the following rules:

§3 All getter methods defined in the library of the administration module must work according to their documentation specified in the available Doxygen documentation (see Appendix D.5).

§4 All setter methods defined in the library of the administration module must work according to their documentation specified in the available Doxygen documentation (see Appendix D.5).

C Document Type Definition for settings.xml

```
<!ELEMENT settings  (hidden?,visible?)>
<!ELEMENT hidden    (boolean|double|integer|string|object|stdobject|enum)*>
<!ELEMENT visible   (boolean|double|integer|string|object|stdobject|enum|menu)*>
<!ELEMENT menu      (boolean|double|integer|string|object|stdobject|enum|menu)+>
<!ELEMENT enum      (element)+>
<!ELEMENT boolean   (#PCDATA)>
<!ELEMENT double    (#PCDATA)>
<!ELEMENT integer   (#PCDATA)>
<!ELEMENT string    (#PCDATA)>
<!ELEMENT object    EMPTY>
<!ELEMENT stdobject EMPTY>
<!ELEMENT element   EMPTY>

<!ATTLIST enum varName CDATA #REQUIRED>
<!ATTLIST enum realName CDATA #IMPLIED>
<!ATTLIST enum desc CDATA #IMPLIED>
<!ATTLIST element realName CDATA #REQUIRED>
<!ATTLIST element value CDATA #REQUIRED>

<!ATTLIST menu title CDATA #REQUIRED>
<!ATTLIST menu desc CDATA #IMPLIED>

<!ATTLIST boolean varName CDATA #REQUIRED>
<!ATTLIST boolean realName CDATA #IMPLIED>
<!ATTLIST boolean desc CDATA #IMPLIED>
<!ATTLIST double varName CDATA #REQUIRED>
<!ATTLIST double realName CDATA #IMPLIED>
<!ATTLIST double desc CDATA #IMPLIED>
<!ATTLIST double min CDATA #IMPLIED>
<!ATTLIST double max CDATA #IMPLIED>
<!ATTLIST integer varName CDATA #REQUIRED>
<!ATTLIST integer realName CDATA #IMPLIED>
<!ATTLIST integer desc CDATA #IMPLIED>
<!ATTLIST integer min CDATA #IMPLIED>
<!ATTLIST integer max CDATA #IMPLIED>
<!ATTLIST string varName CDATA #REQUIRED>
<!ATTLIST string realName CDATA #IMPLIED>
<!ATTLIST string desc CDATA #IMPLIED>
<!ATTLIST string min CDATA #IMPLIED>
<!ATTLIST string max CDATA #IMPLIED>

<!ATTLIST object varName CDATA #REQUIRED>
<!ATTLIST object realName CDATA #IMPLIED>
<!ATTLIST object desc CDATA #IMPLIED>
<!ATTLIST object settingActivity CDATA #REQUIRED>
<!ATTLIST object settingPc CDATA #REQUIRED>
<!ATTLIST object type CDATA #REQUIRED>
<!ATTLIST stdobject varName CDATA #REQUIRED>
<!ATTLIST stdobject realName CDATA #IMPLIED>
<!ATTLIST stdobject desc CDATA #IMPLIED>
<!ATTLIST stdobject type CDATA #REQUIRED>
```

Listing C.1. The DTD for **settings.xml**.

DVD Content

D

In this appendix we give an overview of the content on the enclosed DVD. All paths shown in this appendix are relative to the root of the enclosed DVD.

D.1 Binary

Description: A binary of the GIRAF system. Installing this on an Android device is enough to get started with GIRAF.
Path: `/apk_binary`

D.2 Bibliography

Description: All web-sources from the bibliography list in PDF format.
Path: `/bibliography`

D.3 Code Coverage Analysis Reports

Description: The code coverage analysis reports generated for some of our unit tests.
Path: `/coverage`

D.4 Administration Module Documentation

Description: Contains a series of PDF documentation documents for the administration module. This includes the following titles:

- Getting started with sw6.lib
- Introduction to the settings.xml layout
- Introduction to the sw6.xmlvalidator tool
- Creating Activities for Custom Objects
- Displaying Settings Menu for a Specific Application

Path: `/documentations`

D.5 Administration Module Library Documentation

Description: Documentation of the administration module library generated with Doxygen. To use it, open the `index.html` file.
Path: `/doxygen`

D.6 Multi Project Test Designs, Test Cases, and Test Results

Description: Shows the list of test designs, test cases, and the test results and observations made during the multi project integration test.
Path: `/multi_project_test_results`

D.7 Report

Description: A copy of this report, including LaTeX source code and a compiled PDF.
Path: `/report`

D.8 Validation Tool for settings.xml

Description: This tool is used to validate the content of a `settings.xml` file. Instructions on how to use it are found in Appendix D.4.
Path: `/settings_xml_validator`

D.9 GIRAF Source Code

Description: The source code for all projects related to the GIRAF multi project.
Path: `/source`