AALBORG UNIVERSITY
STUDENT REPORT

Aalborg University

P6 project:

Working On the GIRAF System

Local Database and Category Administration Tool

Group: **sw609f14**
Room: **2.1.12**

Field of study: **Software**

Authors:
Emil S. Jensen
Mathew K. J. Yapp

Supervisor:
Nurefsan Gur

May 28th, 2014

This page is intentionally left blank

Aalborg University
The Faculty of Engineering and Science
School of Information and Communication Technology (SICT)
Selma Lagerlöfsvej 300
9220, Aalborg Ø
Phone number: 99407228
http://www.sict.aau.dk

Title: Working On the GIRAF System
Subject: Local Database and Category Administration Tool
Project period: February 3rd, 2014 - May 28th, 2014
Project group: sw609f14
Writers:

Emil S. Jensen

Mathew K. J. Yapp

Supervisor:
Nurefsan Gur

Copies: 4
Pages: 69 (with appendices)
Pages: 55 (without appendices)
Appendices: 4
Ended on: May 28th, 2014

Abstract:

We describe our contributions to the GIRAF system, using requirements from customers and recommendations from last year.
Through four sprints, we work on two parts of the GIRAF system: LocalDB, a local database, and Cat, a categorizer tool.
In sprint 1, LocalDB is updated to an improved schema developed in 2013, and as part of that cooperate with the other database groups on a description of the databases used.
In sprints 2, 3 and 4, we work on Cat, making it work with the new database schema, as well as improving overall functionality. We also perform a code review and apply the feedback received to improve the quality of the code, as well as perform a functionality test of the application, determining that it behaves almost as expected.

This page is intentionally left blank

Preface

Preface

This report is written by two students from the department of Computer Science at Aalborg University. The students are studying Software Engineering at their 6th semester.

The project has been developed in a multi-project environment, where all groups are on their 6th semester of Software Engineering and have worked on different applications in the system GIRAF. The GIRAF system consists of applications for the Android, IOS and web platforms. GIRAF was started in the spring of 2011, and developed since then by students on their bachelor semester.

This report consists of 8 Chapters. Chapter 1 is a general introduction of the applications we have been working on, Chapters 2 to 5 document the four sprints' work, chapters Chapters 6 and 7 are collaborative writings documenting the databases used, and a code review, and Chapter 8 is the conclusion. If you are working on Cat in 2015 you are advised to read Section 1.2 followed by Sections 8.2 to 8.4 and optionally Chapters 3 to 5.

The contents of this report should be understandable by students, who have completed their 5th semester of Software Engineering at Aalborg University, and have a basic understanding of Android development and SQL.

To ensure readability of the report, a description of some of the most used keywords in the report can be found in Appendix B.

Acknowledgments

We would like to thank Nurefsan Gur, who has been our supervisor this semester, for good and helpful supervision throughout the project.

We would also like to thank Ulrik Nyman for his work as semester coordinator this semester, and inventor of the GIRAF project.

Furthermore we would like to thank project groups sw607f14, sw612f14 and sw615f14 for our collaborate work in the project development, and project groups sw601f14 and sw604f14 for their work on the requirements during sprint 1.

This page is intentionally left blank

Resume

This report shows the work on two parts of the GIRAF system; LocalDB, the local database, and Cat, the "Category Administration Tool". The report starts out with an introduction to the two applications and their dependencies. It then describes the goals for the project, achieved by looking at the requirements made at the start of the project, and the results from a usability test made last year. As the project runs in four sprints, each sprint is set up with its own goals.

Each sprint then has its own chapter, all in the same general form of a set of tasks, sections describing what was done in the sprint and ending with a short reflection.

The chapter on sprint 1 describes the work on LocalDB, rewriting it to a new schema developed last year. This includes several code examples, all concerning a single table to keep a consistent flow.

The chapter on sprint 2 describes work on Cat, starting with a description of the overall system, as a part of the application circumvented the local database and used a XML storage instead. This is followed by a description of the Cat code structure, and how it was changed to use the local database, and how it negatively affected the GUI.

The chapter on sprint 3 describes further work on Cat, starting with a longer list of tasks. This is followed by sections describing how several of these tasks were completed, e.g. how it works with its dependencies and how some functionality was improved.

The chapter on sprint 4 describes the final work on Cat, again starting with a longer list of tasks, of which several are usability improvements. Another of these tasks was to improve the quality of the code. This chapter also contains functionality tests that concludes that Cat works almost as intended, with only a minor flaw.

The two next chapters describes collaborative work. The first concerns the databases in the system, how they are set up, and how they communicate with other applications, and is written in collaboration with two other groups. The second is a code review with another group, with descriptions of what was found, and how the findings was used to improve the code.

The last chapter is the conclusion, with both an overall conclusion, and sub-conclusions for each sprint. This is followed by experiences the project group made, as well as their recommendations for the next group to work on Cat, and GIRAF in general. This is finished by some recommendations for what could be worked on in the future.

The final pages of the report are the bibliography and the appendix, which contains a specification requirement, selected keywords, the old and new database schemata and the results of the functionality tests.

This page is intentionally left blank

Contents

Chapter 1

Project introduction

This project is a "multi-project", meaning that the entire Software 6 semester works together on a single system: GIRAF. GIRAF is an abbreviation of "Graphical Interface Resources for Autistic Folk", and is a system developed on the Android platform for helping people with autism and their guardians [1, p. 3] (Part 1 in the cited report is an introduction to GIRAF and autism). The multiproject part comes into play, as it is a large system, consisting of several different parts, two of which we will work on: LocalDB and Cat, briefly described in the following two sections. The development of the project will be run in a "Scrum of Scrums" like fashion, with weekly meetings, and presentations for the customers at the end of each sprint, of which there will be four.

1.1 LocalDB: local database

LocalDB is the local database in the system. By local, we mean that it stores data locally on a tablet, preventing excessive (and slow) access to the remote database. Oasis, described shortly, acts as a wrapper for the other applications in the system, giving a consistent way of accessing data. We worked on LocalDB during sprint 1.

When we received LocalDB, it was using an outdated schema, shown in Figure C.2, and by the end of sprint 1, it was using the new and improved schema from the Wasteland group from 2013, shown in Figure C.2.

Our work on the local database and our collaboration with other database groups, will all be further described in Chapter 2 and Chapter 6 respectively.

1.2 Cat: Category Administration Tool

Cat is an abbreviation of "Category Administration Tool", with the danish name "Kategoriværktøjet", and is formally speaking designed to make, edit and delete categories, and their subcategories. Furthermore it can create and remove relations between categories, subcategories and pictograms. The Cat tool lets guardians handle categories for a citizen. A category is a set of pictograms available to a citizen, used in other applications in GIRAF such as the application Train. The project CategoryLib acts as a facilitator between Cat and the chosen storage solution. The full set of requirements for Cat can be seen in Appendix A.4. We worked on Cat during sprint 2, 3 and 4.

When we received Cat at the beginning of sprint 2, it was using a local XML storage instead of LocalDB. When the sprint was finished, Cat was using Oasis (and thereby LocalDB) instead, but we had unfortunately broken some features doing that, which depended on the previous storage functionality. This sprint is described in Chapter 3

During sprint 3, we made the GUI of Cat functional again, made debugging easier, and made it work with the applications Launcher and PictoSearch. This is described in Chapter 3.

Sprint 4 was quite productive, with several improvements to the usage of the application, as well as the code, and is described in Chapter 5. We also did a code review with group sw607f14 in this sprint, described in Chapter 7.

1.3 Dependencies

When working with Cat, three other applications is required for it to be functional: Launcher, PictoSearch and Oasis. LocalDB does not have any dependencies, as it is at the lowest level dependency-wise.

1.3.1 Launcher

Launcher (also called GIRAF) is the home screen in the GIRAF project, and is responsible for launching other applications. Its way of doing this has changed between sprint 2, where applications would receive both a guardian and citizen profile, and sprint 4, where applications would only receive a guardian profile.

1.3.2 PictoSearch

PictoSearch provides access to the pictograms in the system, by allowing users to search for them. Cat uses this to allow a guardian to select pictograms.

1.3.3 Oasis

Oasis consists of two parts: OasisLib and OasisApp. OasisLib acts as an access layer to LocalDB. As Cat only uses OasisLib, we will refer to OasisLib as just Oasis. A more technical description of OasisLib can be found in Section 6.3.

1.3.4 GUI

To ensure consistency in the GUI in the applications, the project Giraf-Components contains a set of GUI elements, e.g. buttons, gridviews, and dialogs, which all projects with a GUI can and should use. It uses a consistent naming scheme, with each components type starting with “G”: GButton, GGridView, and GDialog.

1.4 Objectives

The primary objectives of this project will be summarized in this section. The objectives concerning the local database can be found in Appendix A.3, which includes the requirements for the local database found this year. In Appendix A.4, the requirements for Cat can be found. Furthermore the group that worked on Cat last year, found some objectives to be handled this year, described in Section 1.4.1, with a short description of our objectives for each sprint in Section 1.4.2.

1.4.1 Recommendations for Cat from 2013

The group SW605f13 from 2013 worked on Cat and performed usability tests on it. The results of these tests proved that Cat was not as intuitive as intended. This has been documented by Vinther et al. [1, p. 83]. They encountered the following problems:

- The way categories and subcategories were represented in Cat was not intuitive.
  - It was not intuitive which category or subcategory had been selected. This caused the user to be unaware of which category or subcategory the user was placing new pictograms in, or what was in a specific category.
- There was confusion about how to add pictograms.
- The users did not read the help text.
- Only one citizen could be edited (the application should be restarted in order to access another citizen).
- There was confusion about saving.
- The application crashes.

1.4.2 Purpose of each sprint

This subsection describes the objectives and general purpose of each sprint.

Sprint 1

During sprint 1 our goal is to rewrite the existing local database schema to a schema that matched the remote database, such that communication between the two databases and all the relevant data is accessible in the local database (synchronizing the databases will be implemented at a later time).

Sprint 2

The main goal of sprint 2 is to get the application Cat to work with the local database. Before the sprint, Cat relied on XML-serialization to store and load data. Therefore Cat will be reworked, so that its data is accessible and it has access to the database.

Sprint 3

Our goal for sprint 3 is to get the core functionalities of Cat working again, so that it can actually be used as intended.

Sprint 4

The main goal in this last sprint, is to develop features in Cat improving usability, as well as clean up the code and verify that it works as intended

1.5 Regarding screenshots and code examples

Throughout the Cat sprint chapters (Chapters 3 to 5), we show several screenshots of parts of the application. Some of these screenshots (e.g. Figure 5.3) show names. These names are *fictional*, and as such, do not pose a problem regarding privacy.

When reading the code examples, keep in mind that the ones in Chapters 3 and 4 may have been changed in later sprints.

Chapter 2

Sprint 1 - LocalDB

Sprint 1 started on the 24th of February, 2014, and ended on the 19th of March, 2014.

The GIRAF project uses databases in order to store and receive information in different applications of the project. There is a local database, which stores information locally on each device, and a remote database, which stores the information online and makes the same information available on different devices. The connection between LocalDB and the applications using it happens through Oasis. In the past, the database schemata of the local database and the remote database have been different. To make up for this, several projects have revolved around making an application, which should have been able to convert between the schemata. An example of this was the project Morgana, but this solution was not working properly.

2.1 Tasks in sprint 1

Our task this sprint, is therefore to rewrite the local database schema, so that it matches the remote database schema, made by Wasteland in 2013 [2]. This task was found during the planning of sprint 1.

2.2 Key differences in the two schemata

The old database schema from 2012, can be seen in Figure C.2. The new schema, developed by the group Wasteland in 2013, can be seen in Figure C.1. This schema matches the remote database schema. The key differences in the two schemata are the following:

- A table called `pictogram` was introduced, and the table `Media` was removed.
- A table called `category` was introduced.
- Several attributes were changed in different tables, e.g. `profile`, to optimize the database and store more relevant data. An example is that instead of having the attributes `firstname`, `middlename` and `surname`, we now have a single attribute `name`.
- Generally simpler names such as `user` instead of `AuthUsers`.

2.3 Implementation

As described in Section 6.4, SQLite is used as the backend for the local database.

We have implemented LocalDB from the Wasteland schema, but also followed the implementation style of how it was when we received it. For each entity and relationship-set we created a table and corresponding metadata for that table. For a running example of how LocalDB has been updated, the `pictogram` table will be used, starting with the code for creating the table, in Listing 2.1.

```
1  private static final String TABLE_CREATE = "CREATE TABLE "
2    + PictogramMetaData.Table.TABLE_NAME
3      + "("
4      + PictogramMetaData.Table.COLUMN_ID + " INTEGER PRIMARY KEY
           AUTOINCREMENT, "
5      + PictogramMetaData.Table.COLUMN_NAME + " TEXT NOT NULL, "
6      + PictogramMetaData.Table.COLUMN_PUBLIC + " INTEGER NOT NULL, "
7      + PictogramMetaData.Table.COLUMN_IMAGEDATA + " BLOB, "
8      + PictogramMetaData.Table.COLUMN_SOUNDDATA + " BLOB, "
9      + PictogramMetaData.Table.COLUMN_INLINETEXT + " TEXT, "
10     + PictogramMetaData.Table.COLUMN_AUTHOR + " INTEGER, "
11     + "FOREIGN KEY(" + PictogramMetaData.Table.COLUMN_AUTHOR +
12       ") REFERENCES "+ UserMetaData.Table.TABLE_NAME +
13           "(" + UserMetaData.Table.COLUMN_ID + ")"
14     + ");";
```

Listing 2.1: Creating the `pictogram` table.

In Listing 2.1 it is illustrated how the `pictogram` table is created. The primary key of the table is the `_id`, which has type `INTEGER`. When inserting a new `pictogram`, it is not necessary to specify the value of the primary key, as this is done by the `AUTOINCREMENT` statement. The attributes of the `pictogram` table has the types `INTEGER`, `TEXT` or `BLOB`, and has the properties of being nullable or not nullable. In this table we have one foreign key i.e. `author`, which references the `user` table.

In the `pictogram` table, the `PictogramMetaData` is often referenced. We have implemented it in this way to avoid redundancy and to make the tables and code in general more manageable. An example of how we have implemented the metadata class for the `pictogram` table is shown in Listing 2.2, showing how the names of the attributes are stored.

```
1  public class Table implements BaseColumns {
2      public static final String TABLE_NAME = "pictogram";
3
4      public static final String COLUMN_ID = "_id";
5      public static final String COLUMN_NAME = "name";
6      public static final String COLUMN_PUBLIC = "pub";
7      public static final String COLUMN_IMAGEDATA = "image_data";
8      public static final String COLUMN_SOUNDDATA = "sound_data";
9      public static final String COLUMN_INLINETEXT = "inline_text";
10     public static final String COLUMN_AUTHOR = "author";
11 }
```

Listing 2.2: Pictogram Meta Data

2.3.1 CRUD Implementation

We have implemented the four standard persistent storage operations create, read, update and delete, through the methods `insert`, `query`, `update` and `delete`. The implementations of the four methods will be shown with the `pictogram` table as our example, starting with insertion.

Create

Insertions are implemented through the `insert` method in Listing 2.3.

```
1  public Uri insert(Uri uri, ContentValues values) {
2    SQLiteDatabase db = dbHelper.getWritableDatabase();
3    long rowId;
4    Uri _uri;
5
6    switch(sUriMatcher.match(uri)) {
7
8      ...
9
10     case PICTOGRAMS_TYPE_LIST:
11       try {
12         rowId = db.insertOrThrow(PictogramMetaData.Table.TABLE_NAME,
               null, values);
13         _uri = ContentUris.withAppendedId(PictogramMetaData.CONTENT_URI,
               rowId);
14         getContext().getContentResolver().notifyChange(_uri, null);
15       } catch (SQLiteConstraintException e) {
16         _uri = ContentUris.withAppendedId(PictogramMetaData.CONTENT_URI,
               -1);
17       }
18       return _uri;
19
20     ...
21
22   default:
23     throw new IllegalArgumentException("[LocalDB insert] Unknown URI: "
         + uri);
24   }
25 }
```

Listing 2.3: Insert pictogram

The purpose of this method is to insert data into the database. The arguments are used to find the correct table and store the data in that table. The method returns a `Uri` for the new row or rows. In the example above, it is illustrated how a set of `pictograms` can be inserted by the parameter `uri` determining that it is a set of `pictograms`.

Read

Reading is implemented through the `query` method in Listing 2.4.

```
1  public Cursor query(Uri uri, String[] projection, String selection,
       String[] selectionArgs, String sortOrder) {
2    SQLiteQueryBuilder builder = new SQLiteQueryBuilder();
3
4    switch (sUriMatcher.match(uri)) {
5
6      ...
7
8      case PICTOGRAMS_TYPE_LIST:
9        builder.setTables(PictogramMetaData.Table.TABLE_NAME);
10       builder.setProjectionMap(pictogramProjectionMap);
```

```
11        break;
12      case PICTOGRAMS_TYPE_ONE:
13        builder.setTables(PictogramMetaData.Table.TABLE_NAME);
14        builder.setProjectionMap(pictogramProjectionMap);
15        builder.appendWhere(PictogramMetaData.Table.COLUMN_ID + " = " +
              uri.getPathSegments().get(1));
16        break;
17
18      ...
19
20      default:
21        throw new IllegalArgumentException("[LocalDB query] Unknown URI: "
              + uri);
22    }
23
24    SQLiteDatabase db = dbHelper.getReadableDatabase();
25    Cursor queryCursor = builder.query(db, projection, selection,
          selectionArgs, null, null, null);
26    queryCursor.setNotificationUri(getContext().getContentResolver(), uri)
          ;
27    return queryCursor;
28  }
```

Listing 2.4: Query pictogram

The `query` method retrieves data from the database, according to the arguments passed to the method. The result is returned as a `Cursor`. The listing also shows how the database can be queried for either one or several `pictograms`.

Update

The implementation of the update operation is illustrated in Listing 2.5.

```
1  public int update(Uri uri, ContentValues values, String where, String[]
       whereArgs) {
2    SQLiteDatabase db = dbHelper.getWritableDatabase();
3    int rowsUpdated = 0;
4    String rowId;
5
6    switch(sUriMatcher.match(uri)) {
7
8      ...
9
10         case PICTOGRAMS_TYPE_LIST:
11           rowsUpdated = db.update(PictogramMetaData.Table.TABLE_NAME,
                 values, where, whereArgs);
12           break;
13         case PICTOGRAMS_TYPE_ONE:
14           rowId = uri.getPathSegments().get(1);
15           rowsUpdated = db.update(PictogramMetaData.Table.TABLE_NAME,
16                   values,
17                   PictogramMetaData.Table.COLUMN_ID + " = " + rowId +
                         (!TextUtils.isEmpty(where) ? " AND (" + where +
                         ")" : ""),
18                   whereArgs);
19           break;
```

```
20
21      ...
22
23      default:
24        throw new IllegalArgumentException("[LocalDB update] Unknown URI:
              " + uri);
25      }
26
27      getContext().getContentResolver().notifyChange(uri, null);
28      return rowsUpdated;
29    }
30  }
```

Listing 2.5: Update pictogram

The `update` method updates data that is already in the database. It uses the arguments to find the correct table and rows, and use the correct column values, and returns the number of rows that is updated. In our example it is shown how a single `pictogram`, or a list of `pictograms`, can be updated.

Delete

The delete operation is implemented through the `delete` method shown in Listing 2.6.

```
1  public int delete(Uri uri, String where, String[] whereArgs) {
2    SQLiteDatabase db = dbHelper.getWritableDatabase();
3    int rowsDeleted = 0;
4    String rowId;
5    switch (sUriMatcher.match(uri)){
6
7    ...
8
9      case PICTOGRAMS_TYPE_LIST:
10       rowsDeleted = db.delete(PictogramMetaData.Table.TABLE_NAME, where,
             whereArgs);
11       break;
12     case PICTOGRAMS_TYPE_ONE:
13       rowId = uri.getPathSegments().get(1);
14       rowsDeleted = db.delete(PictogramMetaData.Table.TABLE_NAME,
15       PictogramMetaData.Table.COLUMN_ID + " = " + rowId + (TextUtils.
             isEmpty(where) ? "" : " AND (" + where + ")"),
16         whereArgs);
17       break;
18
19   ...
20
21     default:
22       throw new IllegalArgumentException("[LcoalDB delete] Unknown URI:
             " + uri);
23   }
24
25   getContext().getContentResolver().notifyChange(uri, null);
26   return rowsDeleted;
27 }
```

Listing 2.6: Delete pictogram

This method deletes data from the database. It uses the arguments to find the correct table and delete the correct row or rows. It returns the number of rows which where deleted. In the example it is shown how either a list of `pictograms`, or a single `pictogram`, can be deleted.

2.4 Test with the Oasis group

Testing our implementation required using the Oasis group (sw615f14), partly because they had a method for inserting data, partly because we did not have an easy way of verifying our implementation ourselves. Their "Create Dummy Data" (CDD) method was quite straightforward, by inserting different kinds of data into the database. Using this method, a limited black box test was performed. When errors were found (e.g. a missing space in a SQL statement), it was fixed, and CDD was called again.

Continuing in this fashion, to verify that the data had actually been stored, we also queried the database, and verified that the results were as expected.

In this manner, we verified that the SQL statements were correct, and thus that information would be stored when required. We also verified that the information would be stored correctly, and could be retrieved again.

2.5 Reflections on sprint 1

At the end of sprint 1, we had updated the LocalDB schema to use the Wasteland schema, matching the remote database. Thus, our goals for the sprint were completed, although we were not too satisfied with how we verified the implementation.

Chapter 3

Sprint 2 - Cat

Sprint 2 started on the 20th of March, 2014, and ended on the 14th of April, 2014. Our purpose for this sprint is to update Cat to use LocalDB instead of XML for saving data.

3.1 Tasks in sprint 2

- Get to know the code.
- Get Cat to work with the database.
  - Add a category.
  - Add a subcategory.
  - Add a pictogram.
  - Remove a category.
  - Remove a subcategory.
  - Remove a pictogram.

The primary task this sprint is to get Cat to work with the database, which was a requirement from the requirements specification. Other than this, quite some effort will be spent on getting to know the code of Cat and CategoryLib

3.2 Getting to know Cat

In the beginning of sprint 2, we chose to work with the project Cat, of which a short description can be found in Section 1.2. Because Cat is an existing application in the GIRAF project, we will use some time in the start of the sprint, on familiarizing ourselves with the existing code and getting an idea of what needs to be improved in the application. When we started, Cat managed a set of changes, and saved them to XML when needed. Instead we want to save changes when they happen, and use LocalDB as storage.

From this brief analysis, we discovered that Cat and CategoryLib were using XML for saving and loading categories, profiles and their associations. Thus, our goal for this sprint is to update CategoryLib to use LocalDB, through Oasis, and thereby also make Cat use the LocalDB, as intended. In CategoryLib this is done by discarding the code handling the XML files, and changing/adding the methods that handle saving and loading information to use Oasis instead. In Cat, this is done by replacing the old method calls with the new ones in CategoryLib. Unfortunately, saving and loading data in Cat is a bit of a mishmash between using Oasis directly, and using CategoryLib as a wrapper for Oasis.

The illustration in Figure 3.1 shows how the entire GIRAF system, with Cat and CategoryLib in focus, was at the beginning of sprint 2. It shows how CategoryLib, and thereby Cat, used XML files for saving and loading.

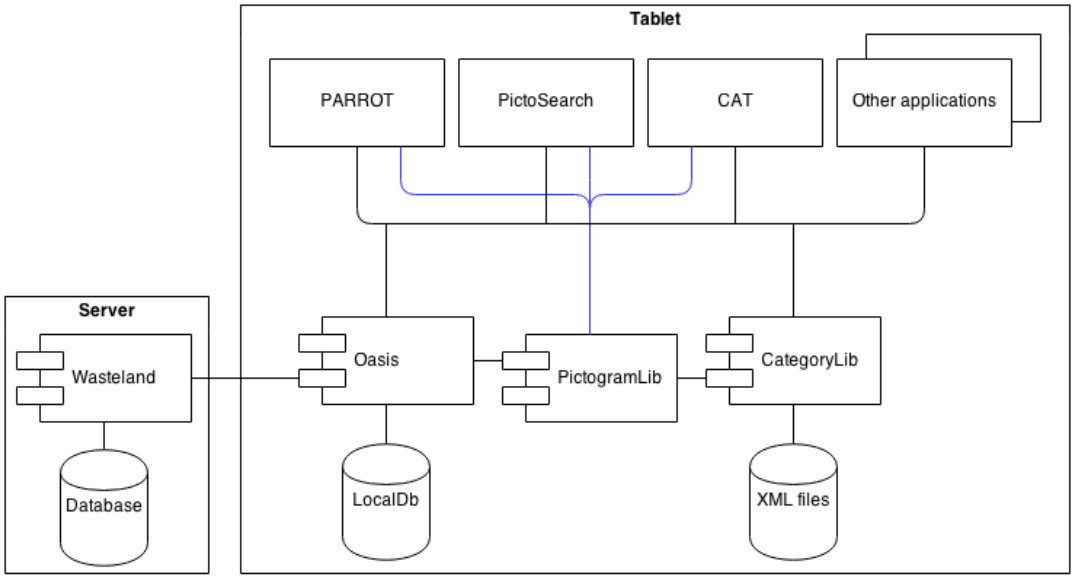

Figure 3.1: The GIRAF system at the start of sprint 2.

Figure 6.1 depicts the current GIRAF system at the end of sprint 2, where the XML-serialization is removed from Cat.

3.3 Structure of Cat

In Figure 3.2 the structure of Cat is illustrated. The UML diagram depicts the different classes in Cat and their relations to each other. This section gives a brief description of each class.

MainActivity This class is the main entry point in Cat, and links the classes in the Cat project together. It is also here that the main functionality of Cat is handled.

CreateCategoryListener and CreateCategoryDialog This interface and class handle the dialog that is shown, when creating a category or subcategory.

IconDialogFragment This class handles when a user wants to change an icon, that can be given to either a category or a subcategory. It is used in the class `SettingDialogFragment`. If the icon setting is chosen, a dialog appears where this can be done, although this functionality did not work when we received the project.

MessageDialogFragment This class handles showing messages to the user, through a dialog. This class should be removed entirely, as Giraf-Components should be used instead.

TitleDialogFragment This class handles the dialog for changing a title, and is used in the class `SettingDialogFragment`.

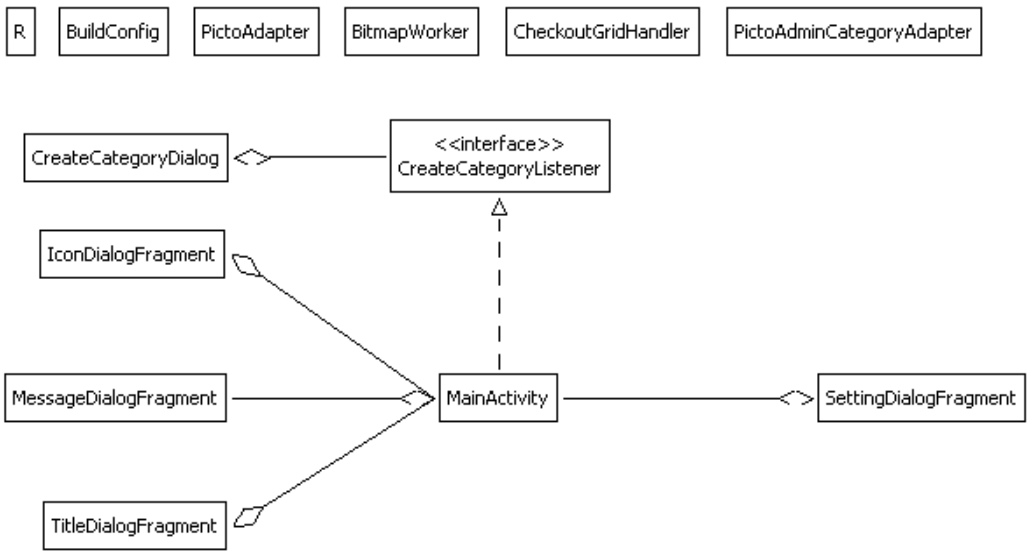

Figure 3.2: UML class diagram of Cat

SettingDialogFragment This class handles the settings option, when long pressing on a category or subcategory. In sprint 2, this involves the settings for title, color and icon.

PictoAdapter This class handles the grid containing pictograms, and their layout.

BitmapWorker This class handles the loading of pictograms in the pictogram grid.

CheckoutGridHandler The purpose of this class is to get all the pictograms in the pictogram grid, and return the ids of all these pictograms in an array of `long`s.

PictoAdminCategoryAdapter This class handles the grid containing categories and subcategories, and their layout.

R and BuildConfig The class `R` contains the definitions for all the resources in the Cat application. The class `BuildConfig` is the build configuration of Cat.

3.4 CategoryLib changes

When we received Cat at the start of sprint 2, the source files for CategoryLib were as shown in Figure 3.3a, and at the end of sprint 2, they are as shown in Figure 3.3b.

As the names in Figure 3.3a suggest, CategoryLib used XML files to store data. Examples of this will not be shown as has been completely removed. As the figures show, all the old files have been removed and replaced with a single file, which now acts as a wrapper for Oasis.

(a) The source files for CategoryLib at the start of sprint 2.

(b) The source files for CategoryLib at the end of sprint 2.

3.4.1 Examples of wrapper functions

As a running example, we will show the methods handling the associations between profiles and categories: adding a category to a profile (Listing 3.1), getting the categories associated with a profile (Listing 3.2), and removing a category from a profile (Listing 3.3).

As can be inferred from these examples, lines 2-4 are "generic", for all methods in CategoryLib, as no parameters may be `null`.

Adding a category to a profile

Listing 3.1 shows the code for adding a category to a profile. If the category to add does not already exist, it is added in lines 4-6, and the connection between the profile and the category is added in lines 10 and 11, returning the return value from the call to Oasis.

```
1  public int addCategoryToProfile(Profile profile, Category category) {
2    if(profile == null || category == null) {
3      throw new NullPointerException("Cannot be null.");
4    }
5
6    if(!categoryControl.getCategories().contains(category)) {
7      categoryControl.insertCategory(category);
8    }
9
10   ProfileCategory profCat = new ProfileCategory(profile.getId(),
         category.getId());
11   int success =  profileCategoryControl.insertProfileCategory(profCat);
12   return success;
13 }
```

Listing 3.1: Method called when adding a category to a profile.

Getting the categories associated with a profile

Listing 3.2 shows the code used for getting the categories directly associated (that is, not subcategories) with a profile. This is done by calling Oasis and returning the value from that call in line 6.

```
1  public List<Category> getCategoriesFromProfile(Profile profile) {
2    if (profile == null) {
3      throw new NullPointerException("Cannot be null.");
4    }
5
6    return categoryControl.getCategoriesByProfileId(profile.getId());
```

```
7 }
```

Listing 3.2: Method called when getting the categories associated with a profile.

Removing a category from a profile

Listing 3.3 shows the method called to remove a category from a profile. This is done in lines 6 and 7, by calling Oasis, and returning what it returned in line 8.

```
1 public int deleteCategoryFromProfile(Profile profile, Category category)
      {
2   if(profile == null || category == null) {
3     throw new NullPointerException("Cannot be null.");
4   }
5
6   ProfileCategory profCat = new ProfileCategory(profile.getId(),
        category.getId());
7   int success = profileCategoryControl.removeProfileCategory(profCat);
8   return success;
9 }
```

Listing 3.3: Method called when removing a category from a profile.

3.4.2 The future of CategoryLib

As the above examples show, CategoryLib is just a wrapper for Oasis. As such, it is a possibility to complete remove CategoryLib, and use Oasis directly.

3.5 Cat changes

The changes to Cat was mostly caused by the changes made to CategoryLib. The notable changes were:

- `PARROTCategory` from CategoryLib was replaced with `Category` from Oasis.
- Where changes were tracked and saved manually before, this is now done automatically through CategoryLib and Oasis.
- Some `long` parameters were changed to `int`, as CategoryLib previously used `long`s for IDs, while Oasis uses `int`s.

3.5.1 Category type change

As CategoryLib previously used `PARROTCategory` to store categories, this had to be changed to the new `Category` type from Oasis. Listing 3.4 shows the signature of the method `updateColor` when we received Cat, and Listing 3.5 shows the signature at the end of sprint 2.

```
1 void updateColor(PARROTCategory category, int pos, boolean isCategory)
```

Listing 3.4: The signature for `updateColor` at the start of sprint 2.

```
1 void updateColor(Category category, int pos, boolean isCategory)
```

Listing 3.5: The signature for `updateColor` at the end of sprint 2.

3.5.2 Managing changes

When we received Cat, it was manually tracked if changes had been made (i.e. anything that could be done in Cat), presumably to only write to the XML file when absolutely necessary. We considered keeping this practice, but as we had updated CategoryLib to use Oasis, changes are now saved automatically. Listing 3.6 shows how the `onPause` method saved changes. By the end of sprint 2, lines 3-22 were no longer needed, and thus removed.

```
1  protected void onPause() {
2    super.onPause();
3    if (subcategoryList != null) {
4      for (PARROTCategory sc : subcategoryList) {
5        if (sc.isChanged()) {
6          sc.getSuperCategory().setChanged(true);
7        }
8      }
9    }
10   if (categoryList != null) {
11     for (PARROTCategory c : categoryList) {
12       if (c.isChanged()) {
13         Log.v("klim", "ready to save");
14         somethingChanged = true;
15         catHelp.saveCategory(c, child.getId());
16       }
17     }
18   }
19
20   if (somethingChanged) {
21     catHelp.saveChangesToXML();
22   }
23 }
```

Listing 3.6: The `onPause` method at the start of sprint 2.

3.5.3 Breaking the GUI

In the last part of sprint 2, we made some ad hoc tests on Cat, to test if the application worked as intended. Through these test we verified that the application was able to store categories and their subcategories in the local database, but discovered that even though the changes were saved, the GUI did not reflect this.

3.6 Reflections on sprint 2

At the end of sprint 2 we had *technically* completed the tasks set in section 3.1, as CategoryLib had been updated to use Oasis instead of XML, and Cat had been updated to use the new CategoryLib. Even though the GUI did not show changes when they were made, it was still stored properly. Some of the tasks for sprint 3 will thus be to make the GUI show changes when they happen.

Chapter 4

Sprint 3 - Cat

Sprint 3 started on the 15th of April 2014, and ended on the 7th of May 2014. Our purpose for this sprint is to make the GUI functional again, and otherwise add features.

4.1 Tasks in sprint 3

With the tasks left over from sprint 2 being few, and relatively simple, we expand our task list with features and improvements, with the intent of finishing some, but not all, of them:

- Show when changes are made (adding/removing categories/subcategories/pictogram).
- Remove relations when an item is removed.
- Ask if the user is sure when deleting an item.
- Make Cat work with Launcher and PictoSearch.
- Global categories - we have only worked with categories associated with a single citizen so far.
- Copy a category to another citizen.
- Copy a subcategory to another category.
- Show which item is selected.
- Make long press work as intended.
- Improve error messages.
- Ensure only Giraf-Components are used.
- Choose a profile directly from Cat.
- Close button.
- Pending icon when not synced.
- Sync button.
- Play sound associated with a pictogram.
- Refactor and clean up the code.
- Possibly remove CategoryLib - it is just a wrapper for Oasis.

Most of these tasks were found, given the requirements specification and the future work from last year. The ones that were not, were the ones regarding copying a subcategory and having global categories. These were found during our weekly status meetings, where we discussed what functionality that would be nice to have in Cat.

4.2 Debugging

To make debugging easier for ourselves, we set up a debugging mode, as seen in Listing 4.1. A previous group had set up some code achieving the same effect, but ours uses Oasis, to get actual data. Using this debug mode means that we can start debugging at once, instead of starting through Launcher (and selecting a citizen), and then attaching the debugger. This is incredibly useful when things are not working in the `onCreate` method.

```
1 if (DEBUG && extras == null) {
2   extras = new Bundle();
3   extras.putLong("currentChildID", proHelp.getChildren().get(0).getId())
        ;
4   extras.putInt("currentGuardianID", proHelp.getGuardians().get(0).getId
        ());
5 }
```

Listing 4.1: Setting up debugging mode.

4.3 Integrating other applications with Cat

An important factor in the GIRAF project is that the different applications works and works together. Cat requires two frontend application for full functionality: Launcher and PictoSearch. If Cat is not started through Launcher, it will not receive the profiles it should work with, and thus be non-functional. If PictoSearch is not installed, it will not be possible to add pictograms to categories, and Cat will be practically useless.

4.3.1 Launcher

Launcher provides the profiles of a citizen and a guardian. It is therefore a necessary application for Cat to work as intended. The way it does this is illustrated in the code segments from the class `MainActivity`, in Listings 4.2 and 4.3.

The code in Listing 4.2 checks if the data received from Launcher is valid. If not, an error dialog is shown, exiting Cat when it is closed. If it is valid, it calls the `getProfiles` method, shown in Listing 4.3, and sets up the GUI. The `getProfiles` method sets the fields `child` and `guardian`, if the data received contains information about these. From here the `child` or `guardian` fields can be used to alter their information regarding categories, subcategories and pictograms, which is the purpose of Cat.

```
1 if(extras == null) {
2   // Code for showing an error dialog, and then exit
3 } else {
4   getProfiles(extras);
5   // Code for setting up GUI
6 }
```

Listing 4.2: Verifying start information.

```
1 private void getProfiles(Bundle extras) {
2   if (extras.containsKey("currentChildID")) {
3     child = proHelp.getProfileById(extras.getInt("currentChildID"));
4   }
5   if (extras.containsKey("currentGuardianID")) {
```

```
6       guardian = proHelp.getProfileById(extras.getInt("currentGuardianID")
            );
7   }
8 }
```

Listing 4.3: Getting profiles.

4.3.2 PictoSearch

As mentioned, PictoSearch gives access to the pictograms in the database. The way PictoSearch is started, is illustrated in Listing 4.4. Here it first loads the variable `request` of type `Intent`, with information such as what application should be started and where the application should be loaded from. The actual activity which is started, is started through the method `startActivityForResult`. If PictoSearch is not installed, an error is shown.

```
1  public void createPictogram(View view) {
2    Intent request = new Intent();
3
4    try {
5      request.setComponent(new ComponentName("dk.aau.cs.giraf.pictosearch"
           , "dk.aau.cs.giraf.pictosearch.PictoAdminMain"));
6      request.putExtra("purpose", "CAT");
7      startActivityForResult(request, RESULT_FIRST_USER);
8    } catch (Exception e) {
9      message = new MessageDialogFragment(R.string.search_missing,this);
10     message.show(getFragmentManager(), "notInstalled");
11   }
12 }
```

Listing 4.4: Starting PictoSearch

4.4 Removal of items in Cat

In this sprint, the functionality of Cat is improved, so changes are shown when they are made.

4.4.1 Removing a category

Removing a category in Cat involves removing the actual category from the database, along with its associated subcategories and pictograms. This is done first in the database, and then in the GUI. The code handling this is shown in Listing 4.5.

```
1  public void askDeleteCategory(View view) {
2    GDialogMessage deleteDialog = new GDialogMessage(this,
3      getString(R.string.confirm_delete),
4      new View.OnClickListener() {
5        @Override
6        public void onClick(View view) {
7          catlibhelp.deleteCategory(selectedCategory);
8          categoryList.remove(selectedLocation);
9          selectedCategory = null;
10         categoryGrid.setAdapter(new PictoAdminCategoryAdapter(
               categoryList, view.getContext()));
```

```
            selectedSubCategory = null;
            subcategoryList.clear();
            pictograms.clear();
            subcategoryGrid.setAdapter(new PictoAdminCategoryAdapter(
                subcategoryList, view.getContext()));
            pictogramGrid.setAdapter(new PictoAdapter(pictograms, view.
                getContext()));
            updateButtonVisibility(null);
        }
    });
    deleteDialog.show();
}
```

Listing 4.5: Removing a category.

4.4.2 Removing a subcategory

Removing a subcategory is similar to removing a category, except for the fact that the associated supercategory to the subcategory should not be removed. This means that the subcategory and its associated pictograms are removed with this action. The code handling this, is shown in Listing 4.6.

```
public void askDeleteSubCategory(View view) {
    GDialogMessage deleteDialog = new GDialogMessage(this,
        getString(R.string.confirm_delete),
        new View.OnClickListener() {
            @Override
            public void onClick(View view){
                catlibhelp.deleteCategory(selectedSubCategory);
                subcategoryList.remove(selectedLocation);
                selectedSubCategory = null;
                pictograms.clear();
                subcategoryGrid.setAdapter(new PictoAdminCategoryAdapter(
                    subcategoryList, view.getContext()));
                pictogramGrid.setAdapter(new PictoAdapter(pictograms, view.
                    getContext()));
                updateButtonVisibility(null);
            }
        });
    deleteDialog.show();
}
```

Listing 4.6: Removing a subcategory.

4.4.3 Removing a pictogram

Removing a pictogram does not affect associated categories or subcategories. This action therefore 'only' removes the selected pictogram. The corresponding code is shown in Listing 4.7.

```
public void askDeletePictogram(View view) {
    GDialog deleteDialog = new GDialogMessage(view.getContext(), R.
        drawable.content_discard, getString(R.string.confirm_delete), "",
        new View.OnClickListener(){
```

```
3       @Override
4       public void onClick(View v) {
5         if(selectedSubCategory == null) {
6           catlibhelp.deletePictogramFromCategory(selectedPictogram,
                selectedCategory);
7         } else {
8           catlibhelp.deletePictogramFromCategory(selectedPictogram,
                selectedSubCategory);
9         }
10        pictograms.remove(selectedLocation);
11        selectedPictogram = null;
12        pictogramGrid.setAdapter(new PictoAdapter(pictograms, v.getContext
              ()));
13      }
14    });
15    deleteDialog.show();
16  }
```

Listing 4.7: Removing a pictogram.

4.4.4 "Accept deletion" dialog

When removing an item in Cat, a dialog box appears, to ensure that the user actually intended to delete this item and did not make a mistake. This dialog box is depicted in Figure 4.1. If the user presses "Godkend" (accept), the item is removed, and if the user presses "Fortryd" (cancel), nothing is deleted.

Figure 4.1: Dialogbox for deleting an item.

4.5 Close button

To fulfill the requirement that all applications in GIRAF should have a close button, we implemented a close button, to make it easier for the user to see where the application should be closed. The close button in Cat is placed in the top right corner, and is illustrated in Figure 4.2.

Figure 4.2: Close button.

4.6 Settings

When long-pressing on a category or a subcategory, a dialog is issued, containing various settings. In sprint 3 these settings are the following:

- Change title.
- Change color.
- Change icon.

We implemented the settings 'Change title' and 'Change color' in sprint 3, for both categories and subcategories.

One of the other requirements for Cat is that a guardian is able to copy categories between citizen profiles, and this should also be implemented in the settings dialog. We therefore also want to implement this. Both the implementation of the setting 'Change icon' and copying a category are described in Chapter 5.

4.7 Reflections on sprint 3

We completed several tasks during sprint 3, most notably the GUI now show changes as they are made. To itemize:

- The GUI now updates when changes are made.
- Relations are now removed when categories or subcategories are removed.
- When the user tries to delete an item, it is asked if s/he is sure.
- Cat works with the current versions of Launcher and PictoSearch.
- We have worked on the long-press dialog corresponding to settings, and parts of it work.
- We use more Giraf-Components.
- A close button has been added.

The tasks we did not complete will be looked at in sprint 4.

One thing that was incredibly useful was the debug mode we set up. Apart from that, we are quite satisfied with the improvements to the GUI (especially that it now works), and how the overall application got a more "GIRAF"-y feel.

Chapter 5

Sprint 4 - Cat

Sprint 4 started on the 8th of May 2014, and ended on the 20th of May, 2014. Our purpose this sprint is to improve and add functionality, and clean up the code.

5.1 Tasks in sprint 4

In this, the last, sprint, we will continue working with the tasks from sprint 3, this time in a weakly prioritized list:

1. Copy a category to another citizen.
2. Show which item is selected.
3. Improved error message.
4. Make settings work as intended.
5. Give a category or a subcategory an icon.
6. Edit the icon on a category or a subcategory.
7. Choose a profile directly from Cat.
8. Refactor and clean up the code.
9. Resolve the confusion with saving discovered last year, described in Section 1.4.1.
10. Ensure only Giraf-Components are used.
11. Global categories - we have only worked with categories associated with a single citizen so far.
12. Copy a subcategory to another category.
13. Pending icon when not synced.
14. Sync button.
15. Play sound associated with pictogram.
16. Possibly remove CategoryLib - it is just a wrapper for Oasis.

These tasks were the ones not completed in sprint 3, except for the one involving icon. This task was found during a weekly status meeting, and by a discussion with group sw615f14 who works on an application (Piktooplæser) that uses Cat.

We will focus on the first 9 points, with some focus on point 10, but not too much, as some parts would require major reworking to be functional if we did.

5.2 Showing when debugging

To make it easier to see when Cat is started in debug mode, a label has been added at the top of the application, shown in Figure 5.1. The label is located just to the right of the name of the active citizen, and is only visible when started in the debug mode in Section 4.2. The code handling debug mode is shown in Listing 5.1

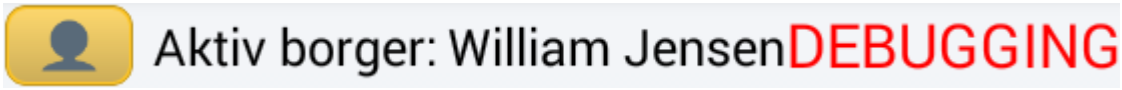

Figure 5.1: The application when started in debug mode.

```
1  private Bundle setupDebug() {
2    Bundle extras = new Bundle();
3
4    Profile guard = profileController.getGuardians().get(0);
5    Profile kid = profileController.getChildrenByGuardian(guard).get(0);
6
7    extras.putInt("currentChildID", kid.getId());
8    extras.putInt("currentGuardianID", guard.getId());
9
10   // Show that we are debugging
11   TextView debugText = (TextView) this.findViewById(R.id.DebugText);
12   debugText.setVisibility(View.VISIBLE);
13
14   return extras;
15 }
```

Listing 5.1: Code handling debug mode.

5.3 Usability improvements

A few things have been done to improve the usability of the application: a "change user" button is added, a description of the autosaving happening is added, and it is shown which item was pressed.

5.3.1 Change user button

One change that has been made to the GUI is the addition of a button to change the active user. Figure 5.2 shows the application before the button was added, and Figure 5.3 shows the application with the button added. In both cases, the figures shows the top-left corner of Cat. They also show that the label "Aktivt barn" (Active/selected child) has been changed to "Aktiv borger" (Active/selected citizen).

5.3.2 Telling the users that changes are saved automatically

In the bottom right corner of the application, a label has been added telling that changes are saved automatically, thus handling regarding saving. This is shown in Figure 5.4.

Figure 5.2: The application before the addition of the button.

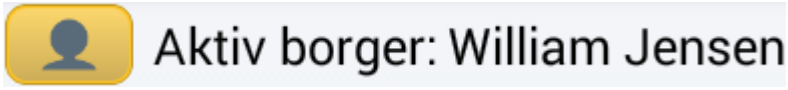

Figure 5.3: The application after the addition of the button.

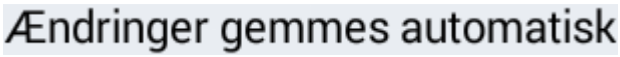

Figure 5.4: Info-label regarding changes.

5.3.3 Showing which item has been selected

Group sw609f13 who worked on Cat last year, discovered through usability tests (Section 1.4.1) that it was not intuitive which item had been selected. Therefore we made it a priority to make this more intuitive, to help the users. We considered several ways of achieving this, e.g. marking it somehow, or showing an animation, but decided on the animation, as this was easiest to implement, to indicate when an item is selected. This animation is fairly simple and just makes the selected item 'pop out'. The code for this animation can be seen in Listing 5.2. This animation is started every time an item such as a category, a subcategory or a pictogram is pressed. The code for starting the animation is illustrated in Listing 5.3.

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <scale android:fromXScale="0.8" android:fromYScale="0.8"
        android:toXScale="1.1" android:toYScale="1.1"
        android:pivotX="50%" android:pivotY="50%" />
</set>
```

Listing 5.2: Select item animation

```
private void setupOnClickActions(View v, int position, int id){
  v.startAnimation(AnimationUtils.loadAnimation(v.getContext(), R.anim.
      pop));
  updateSelected(v, position, id);
  updateButtonVisibility(v);
}
```

Listing 5.3: Starting the animation

5.4 Ensuring a citizen is selected at start

A change to Launcher made it necessary to manually select a citizen when launching Cat. Because we were not sure that Launcher would never send a citizen profile, we handled both the case where it does, and the case where it does not, the latter shown in Listing 5.4.

```
private void selectAndLoadChild() {
  if (child == null || child.getId() == -1) {
```

```
3       final GProfileSelector selector = new GProfileSelector(this,
            guardian, null, false);
4       selector.setOnListItemClick(new AdapterView.OnItemClickListener() {
5         @Override
6         public void onItemClick(AdapterView<?> adapterView, View view, int
              i, long l) {
7           child = helper.profilesHelper.getProfileById((int) l);
8           loadChildAndSetupGUI();
9           selector.dismiss();
10        }
11      });
12      try {
13        selector.backgroundCancelsDialog(false);
14      } catch (Exception e) {
15        finish();
16      }
17      selector.show();
18    } else {
19      loadChildAndSetupGUI();
20    }
21  }
```

Listing 5.4: Ensuring a citizen is selected

5.5 Icon changes

Several changes have been made in Cat associated with its (sub)category icon functionalities.

To get pictograms which can also be used for icons, we use PictoSearch, described in Section 1.3.2. The key method when receiving pictograms is `OnActivityResult`, found in `MainActivity`. This method is called when a result from PictoSearch is present. The method for calling and starting PictoSearch is shown in Listing 4.4. It handles four cases regarding pictograms, which are the following:

- Adding pictograms to a category or a subcategory.
- Adding an icon to a category or a subcategory.
- Changing an icon on a category.
- Changing an icon on a subcategory.

The methods functionality regarding icons will now be described.

5.5.1 Adding an icon to a category or subcategory

To differentiate between the four cases the parameter `requestCode` of type `int` is used in a switch statement. In the code for adding an icon, it is shown how the global variable `newCategoryIcon` of type `Pictogram` is set to the pictogram which is chosen as an icon, if there is one or more results from PictoSearch. The code for this is illustrated in listing 5.5.

```
1  if (checkoutIds.length >= 1) {
2    newCategoryIcon = helper.pictogramHelper.getPictogramById(checkoutIds
         [0]);
3    if(checkoutIds.length>1)
4    {
5      alertDialog(this, "Kun et pictogram kan vaelges som ikon til
           kategorien.", "Det oeverste i listen er valgt.");
```

```
6     }
7 }
```

Listing 5.5: Getting the icon to a set on a category or subcategory.

If more than one pictogram is selected, the dialog in Figure 5.5, telling that only one pictogram can be used as an icon, is shown, and the first pictogram selected.

Figure 5.5: Alert dialog, when the user added more than one pictogram as icon.

5.5.2 Changing an icon in a category or subcategory

When changing an icon, the class `SettingDialogFragment` is used, which contains all settings and is triggered by a long press. The code in Listing 5.6 is a segment from the method `onCreateDialog`. This method handles the different settings options. The 'Icon' setting first starts PictoSearch corresponding to how it is done when creating a pictogram. The code for this is shown in Listing 4.4. The difference from when creating a pictogram is the code shown in Listing 5.6. This code checks whether it is a category or subcategory. If it is a category, it calls the method `startActivityForResult` with a request and the value 3. This in turn calls the `OnActivityResult`, when a result from PictoSearch is received. The same happens if it is a subcategory, except from the fact that the value passed is 4. Thereby the differentiation between a category and a subcategory is handled. In the method `OnActivityResult` these values corresponds to the parameter `requestCode`, as described above. The method `getActivity`, gets the activity and in our case ensures that the result is sent to `MainActivity`.

```
1 if (isCategory == true) {
2   getActivity().startActivityForResult(request, 3); //Sends the info to
        OnActivityResult in MainActivity. Category
3 } else {
4   getActivity().startActivityForResult(request, 4); //Sends the info to
        OnActivityResult in MainActivity. Subcategory
5 }
```

Listing 5.6: Starting PictoSearch.

5.5.3 GUI changes regarding icon

Several changes have been made to the GUI as well, when introducing icons on categories and subcategories. When creating a category or subcategory another button called 'Ikon'

has been added in the dialog box. When pressing on this, an icon can be added to the given category or subcategory.

Another GUI change in this dialogbox, is the use of the GIRAF colors on the buttons to enhance consistency. The appearance of the dialogbox before can be seen in Figure 5.6a, and the appearance of the dialogbox now can be seen in the Figure 5.6b.

(a) Creation of category before.

(b) Creation of category now.

5.6 Copying

Copying a category is quite straightforward. It does the following (the code is not shown, as it is twice as long as this enumeration):

1. Setup Oasis controllers.
2. Get the citizen to copy to.
3. If the citizen to copy to is the same as the one copying from, show an error and stop.
4. If the citizen to copy to has a category with the same name as the one being copied, show an error and stop.
5. Copy the selected category to the other user.
6. Copy all subcategories on the selected category to the new category on the other user, while counting how many are copied.
   (a) Copy all pictograms on all subcategories to the respective subcategory on the other user, while counting how many are copied.
7. Copy all pictograms on the selected category to the category on the other user, while counting how many were copied.
8. Show a message telling how much was copied.

5.7 Refactoring

A part of this sprint was spent refactoring the code, and prepare it for the next year, to allow an easier introduction to the code than we had.

5.7.1 MainActivity

The `onCreate` method in `MainActivity` started out as a longer method spanning around 119 lines, containing a general setup (loading profiles from the Launcher, setting up listeners on the GUI), enabling "debug mode" if needed, and verifying that profiles was actually retrieved, in the following order:

- Initialize global variables.
- Setup debug mode if needed.
- Verify profiles was sent.
- Load profiles (by actually calling a method doing that).
- Setup GUI and listeners.

After cleaning it up, it has become a nice little method spanning around 21 lines, doing the following:

- Setup debug, by calling a method handling that.
- Verify profiles was sent, and call a method to handle if they were not.
- Load profiles by calling the same method as earlier.
- Call a method handling if a citizen was not selected from the launcher.
- Call a method loading the citizens categories.
- Call a method setting up the GUI and listeners.

The initialization of global variables has been removed completely from the method, by doing it at declaration time.

5.7.2 SettingsDialogFragment

The constructor in `SettingsDialogFragment` was originally around 50 lines, and clean enough to be easily understandable. Adding the ability to copy categories was quite long however, at around 80 lines, so this was refactored out to its own method. Due to the way we count how much has been copied, it was not easily possible to refactor this method shorter. It is however, quite straightforward, so it should be easy enough to understand what it does.

5.7.3 Class diagram after refactoring

Refactoring the code in Cat, some classes have been removed, and further changes have been made to the overall structure. The removed classes are:

- `CheckoutGridHandler`.
- `MessageDialogFragment`.
- `IconDialogFragment`.

The class `CheckoutGridHandler` has been removed, because it was not used anymore, as we use the database instead of XML-serialization to load and save pictograms. In the same vein, `MessageDialogFragment` has also been removed, as we use Giraf-Components to issue messages to the user. This was done to have consistency among the applications in GIRAF.

The functionality of `IconDialogFragment` has been moved to the classes `MainActivity` and `SettinsDialogFragment`. Furthermore we also used the application PictoSearch for providing icons, instead of the one provided by the IconDialogFragment.

After fixing the GUI in Cat including the classes `PictoAdapter` and `PictoAdminCategoryAdapter`, there has become aggregation on these from `MainActivity`.

All these changes are represented of the class diagram of the current Cat application in Figure 5.7.

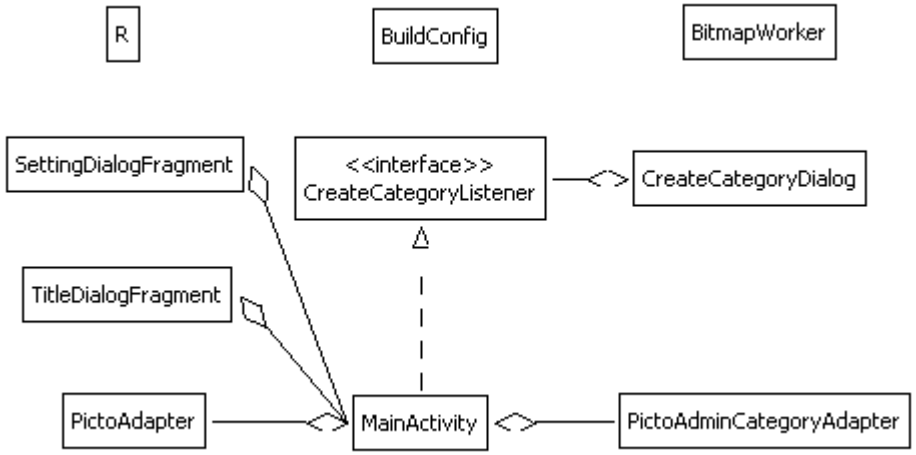

Figure 5.7: Class diagram of Cat by the end of sprint 4.

5.8 Test

To ensure that Cat works as intended, we have made some functionality tests on our final version of Cat. These tests can be found in Appendix D. Overall the application performs as expected. There are only two cases, regarding the same issue, that show that Cat does not perform entirely as expected:

Edit dialog, change title on category, with empty title.
Expected: Error.
Result: Category now has an empty title.
Edit dialog, change titel on a subcategory, with empty title.
Expected: Error.
Result: Subcategory now has an empty title.

The undesirable results occur when changing the title of either a category or a subcategory, and leaving the title blank. This can easily be fixed by a simple check on whether or not the given title is an empty string. If the check shows that the given title is indeed empty, an error message should be issued. This flaw does not make the application crash or anything of that sort, but it is not intuitive to have a category or subcategory without a title.

All in all the tests show that Cat behaves almost as intended, and does what is expected of it.

5.9 Reflections on sprint 4

During sprint 4, we completed all the tasks that we had focused on, as described in Section 5.1. These are shown in the following:

- Copy a category to another citizen.
- Show which item is selected.
- Improved error message.

- Make settings work as intended.
- Give a category or a subcategory an icon.
- Edit the icon on a category or a subcategory.
- Choose a profile directly from Cat.
- Refactor and clean up the code.
- Resolve the confusion with saving discovered last year, described in Section 1.4.1.
- (Ensure only Giraf-Components are used.)

Furthermore we performed functionality test on the functionality of Cat, and thereby verified that it worked as intended.

Because we did not have to familiarize us with new code and had clear and manageable objectives in this sprint, we completed what we wanted to, which were quite satisfying. It was also nice to see Cat grow even more complete, with the improved functionality.

Chapter 6

Sprint 1 collaborative writing: database

This chapter is written in collaboration with sw609f14, sw612f14, and sw615f14.

6.1 The Structure of the Databases

The databases of the GIRAF environment is structured as shown in figure 6.1. The main database is Remote DB, which each device synchronizes against. This enables users to log in at any device an have up-to-date information about himself. The local database on each device is LocalDB, which applications use to store and retrieve information from. It is the task of Sync-lib to facilitate synchronization between the Remote DB and the Local DB, which can be initiated from a specific App, illustrated as App_{Sync}. This construction has been created to ensure that the GIRAF environment works even if the device is offline. The synchronization should be automatic.

To communicate with the local database, the applications in the GIRAF environment use the Oasis-library, which is an API to the local database. This has been created to ease the programming for the applications of GIRAF.

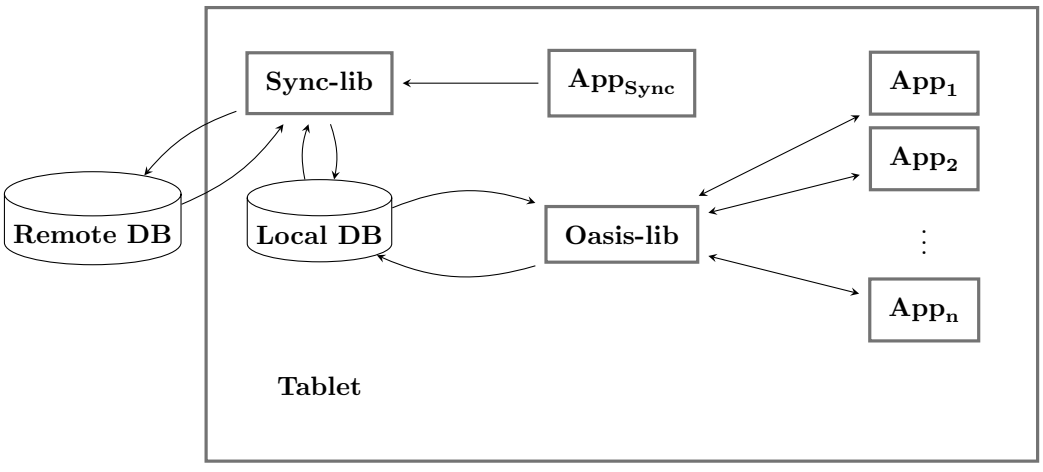

Figure 6.1: Overview of the Database Structure

6.2 Remote database

The purpose of the remote database is to act as a data-central from where clients can synchronize their local database by sending and receiving updated content.

Database Management Systems

The system is currently running on MySQL, which is adequate for our purposes at the moment. However, if it is found that MySQL does not support needed features, another DBMS, such as PostgreSQL or Oracle, could be considered. It should be noted that Oracle is not free to use.

The Schema

The existing schema was created by students from previous semesters at Aalborg University [2]. The schema is in Boyce-Codd normal form, and consists of a total of 16 tables, where nine of these are handling the relations between the seven remaining tables. These seven primary tables are, in no particular order, Pictogram, Category, Department, Profile, User, Application and Tag. The different relations is explained in the following section.

Pictogram

Pictogram stores image data with associated sounds and descriptions.

- Every *Pictogram* can have associated with it an author, which is the administrative *User* who created it.
- A many-to-many relation to *Tag* exists through the relation-table **pictogram_tag**.

Category

Category is used to group pictograms.

- A *Category* can be a child of another *Category*, specified by the foreign key **super_category_id**.
- A many-to-many relation to *Pictogram* exists through the relation-table **pictogram_category**.

Department

Department is used to group profiles by affiliation.

- A *Department* can be a child of another *Department*, specified by the foreign key **super_department_id**.
- Every *Department* can have associated with it an author, which is the administrative *User* who created it.
- A many-to-many relation to *Pictogram* exists through the relation-table **department_pictogram**.
- A many-to-many relation to *Application* exists through the relation-table **department_application**.

Profile

Profile stores relevant information about persons, such as contact information and roles.

- A *Profile* is connected to a specific *Department* through the foreign key **department_id**.
- A *Profile* can be associated with a *User* through the foreign key **user_id**.
- Every *Profile* can have associated with it an author, which is the administrative *User* who created it.
- A many-to-many relation to *Pictogram* exists through the relation-table **profile_pictogram**.
- A many-to-many relation to *Category* exists through the relation-table **profile_category**.
- A many-to-many relation to itself exists through the relation-table **guardian_of**.

User

User stores login information and is used to gain access to the system.

- A many-to-many relation to *Department* exists through the relation-table **admin_of**.

Application

Application stores meta information about applications and is used to control access to applications.

- Every *Application* can have associated with it an author, which is the administrative *User* who created it.

Tag

Tag is used to ease searching for pictograms.

6.3 Structure of Oasis Lib

The structure of Oasis consists of models, controllers and metadata. The responsibility of the models and controllers is determined by following the Model-View-Controller pattern.

Model

The model describes the structure and handles the logic of the system. Change of data is handled in the model, and in case of major changes of data, the view is updated to display the current data.

View

The view displays the data, and it is used to interact with the system. The view interacts with the controller to request existing data and to modify or create new data.

Controller

The controller is the communication medium between the model and the view. It is only possible to access and modify data through the controller.

Implementation

In Oasis there are only models and controllers. Oasis is used to communicate with the database, and therefore it has no graphical interface and hence no view. The views are represented by each application in the GIRAF environment. Metadata in Oasis is used to specify the information about the database, and it is not a part of the Model-View-Controller pattern.

Figure 6.2 shows the design of our implementation.

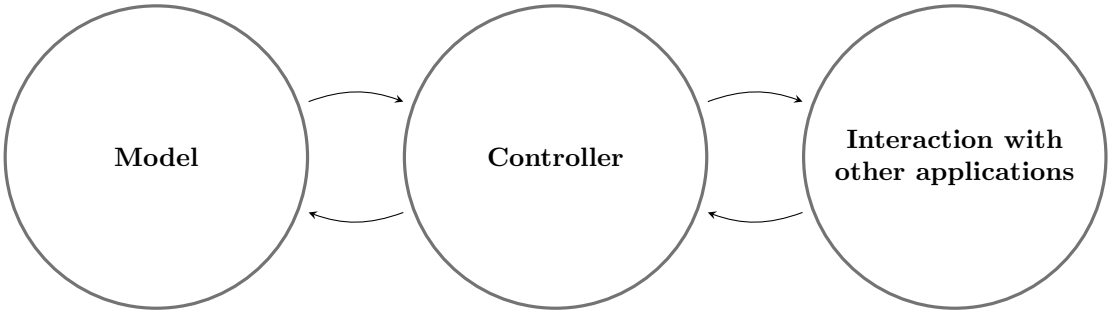

Figure 6.2: The structure of Oasis lib

6.4 Local database

As the applications run on the Android system, an obvious choice of database is SQLite, as it is fully supported on Android [3]. SQLite was also used by the previous database groups. An excellent tutorial for using SQlite is written by Vogel [4], describing the use of both SQLite and `ContentProviders`.

SQLite

SQLite is a public domain, light-weight, embedded SQL database engine [5]. It implements most of SQL-92 [6], but omits a few, for this project irrelevant, features [7].

Implementation

The implementation of LocalDB follows the recommended approach for creating an SQLite database [3]. For other apps (Oasis only) to use the database, a class extending `DbProvider` [8] was implemented, giving access to the standard CRUD operations.

Chapter 7

Sprint 4 collaborative writing: code review

For a collaborative part, it was chosen to perform a code review between sw609f14 (Cat) and sw607f14 ('Sekvens'). According to Wikipedia [9], code reviews can be done in several different ways, e.g. pair programming or informal walkthroughs. We adopted recommendations 2, 3, 4, and 7 from Cohen's practices [10]. We also applied the way "Foredecker" meet up again after reading through the code and discuss it [11]. Our primary focus was to keep it not too long, as to be able to maintain focus, as Cohen suggests in point 3 in his article.

Our goal with the code review was to get other people to look through our code, and find obvious errors the groups had stared themselves blinds on, as well as the general quality and readability of the source code, and if the structure of the source code - methods, classes, names - was meaningful.

Group sw607f14's finding in Cat's code are listed in section 7.2, and group sw609f14's findings in 'Sekvens' are listed in section 7.3. How the feedback was used, is described in section 7.4 for 'Sekvens', and section 7.5 for Cat.

7.1 Common traits

It was found that the code in general had good readability. Both method and variable names made sense in the context. The code was generally split up well into proper methods. The code was well commented, explaining components that were not self-explanatory.

7.2 'Sekvens': looking through Cat's code

Duplicated code Throughout the code certain variables were cleared several times, creating some redundant code. It was found that a better alternative could be to create a method to clear these variables rather than doing it manually every time. The same applied to updating some of the graphical components. These updates could be done in a single method rather than several times throughout the code.

Lazy class A minor issue was found that Cat overrode Android's `onPause` method. The only action taken within this method was however to call Android's super `onPause`, meaning

that the overridden `onPause` did nothing that would not have been done automatically in the Android lifecycle.

Contrived complexity / Complex conditionals In a few cases, unnecessary checks were performed. One case was an if-sentence that checked if a variable was empty. If it was indeed empty, it would recreate that variable as empty.

Another case was excessive use of if-sentences where the checks could either be combined or alternatively be set up using a switch. While the code is generally easily readable, this could give some of the code even better readability.

Another case of contrived complexity was assigning all properties of an element to another, rather than copying the entire element. A case was also found where an instantiated variable was instantiated again for no apparent reason.

Other A few examples were found where the code was perfectly functional, but technically wrong, not used as intended or unnecessary. An example of wrongly used code was the use of controllers. Rather than using the helper class from Oasis, these helpers were circumvented by instantiating new controllers.

Another minor detail was to cancel a window rather than dismissing it. While not changing any functionality, it would have been technically correct to use Android's dismiss method.

Adapters were used a lot in the Activity and were fully functional. In the code however, it turned out that every time the data for an adapter was changed, a new adapter was created on that data. The correct way would have been to use Android's `notifyDataSetChanged` method to update the existing adapter.

Lastly a few potential bugs were found. Some images were not centered as intended and it was possible to avoid selecting a child by using the Back button, which would create further issues in the application.

7.3 Cat: looking through 'Sekvens's code

Well structured and good use of methods Generally the code made and edited by group sw607f14 were well structured and simple to read and understand. It had a good and appropriate use of methods, which made it easier to get an overview of the class. Furthermore these methods were of appropriate size.

Grouping of variables Variables and their declaration could have been grouped better, such that related variables were placed together. This way it would have been easier to find certain variables, when trying to find a variable.

Nested classes with some inappropriate intimacy The class `MainActivity` had two nested classes, which depended on variables from `MainActivity`. These could be moved into separate files and made independent from the `MainActivity` to avoid inappropriate intimacy among the classes.

Some excessive use of literals instead of variables When accessing other applications through Zebra, hardcoded literals where used to send data to the starting application. These literals could be made into variables, such that multiple equivalent literals could be avoided.

7.4 'Sekvens': applying feedback from Cat

As a result of the Cat groups constructive feedback, the `MainActivity` of 'Sekvens' was changed.

Moving code to XML Some code was highlighted which could be set in XML rather than be done programmatically during runtime, and we did exactly that, as it were preferred setting elements in XML whenever possible.

Try-Catch It was noticed by Cat that several `try-catch` were present without any real functionality; Errors would be caught but not dealt with. Most of the `try-catch` present in the code was not needed at that point, and were simply removed as it was not possible for the exceptions to occur.

Multiple classes within same file As we had two other classes in the same file as the `MainActivity` class, it was suggested to either separate them from the file or move them to the bottom of the file to separate them from methods used in `MainActivity`. Since the classes were minor and uniquely used in `MainActivity`, it was chosen to move the classes to the bottom of the file as suggested.

String literals Rather than using constants when sending or receiving intents, `MainActivity` had string literals, which Cat noted is a bad idea. This change was not completed in time, but is definitely a good idea.

Rejected suggestions Feedback on code style was also given. This included using `foreach` loops rather than using counters with `for` loops and sorting variables by functionality rather than type. It was also suggested to refactor a case where a method was nested within another. This was however also rejected, as we found that for the particular case, increased readability was better than moving the inner method.

7.5 Cat: applying feedback from 'Sekvens'

Several changes were made to the class `MainActivity` as a result of the feedback from sw607f14.

Adapters In several places we created and set new adapters to update the GUI. This was changed to use the `notifyDataSetChanged` method instead. In some places however, this did not work, and we were thus forced to set the adapters again.

Controllers Instead of instantiating our own instances of the necessary controllers, we switched to use the `Helper` class from Oasis. This made some lines a bit longer, but was otherwise worth it. In the process, we also removed all usage of CategoryLib in `MainActivity`.

Dialogs Instead of calling `cancel` on dialogs when they were closed, we now call `dismiss`.

onPause method As the `onPause` method was unneeded, it was removed. It had been used before when Cat was using XML files to store its data, but only the code handling this had been removed.

Contrived complexity and complex conditionals As was noted, we had some cases of complex conditionals. The `OnActivityResult` method was the worst in this. Before refactoring the method, it consisted of several if-statements (some of which made a "backwards" check, e.g. `if (isIcon != true)` { `...` }). After refactoring, the method contained a switch statement, with four cases, which is much easier to read.

Rejected suggestions The potential bug where it was possible to avoid selecting a child by using the Back button, was a general issue involving the Giraf-Component `GProfileSelector`, and can best be fixed by the group working on GUI. Therefore it was unfortunately not handled this year, but should be looked at next year.

Other suggestions for improving the code was rejected simply due to a lack of time.

Chapter 8

Conclusion

As we have not worked in a multiproject before, it has been a new experience, with some things learned the hard way: communication is *important*, especially when applications depend on each other. Regardless of the difficulties that comes with doing something new, it has overall been a good experience, where we have learned much on how bigger projects are developed. It has also been very motivating to have people who will use the software after completion, contrary to previous projects which most likely are never seen again after the exam.

Through our work in the multiproject, we have experienced a lot of work with other groups than our own, which have been very different from what we generally have been used to, during earlier semesters. This have had both good and bad sides. We experienced a lot of door-to-door communication, to for instance fix simple tasks or check that something was working. Because of the substantial amount of this, it was hard to document every little change that happened during the project.

To ensure that every group was on the right track, we had brief weekly status meetings, where each group gave a short summary of what they had been doing and what they were going to do, and discussed if any problems had occurred.

Likewise we also had sprint evaluations by the end of each sprint, which also showed the status of the group, but more importantly also showed the products to the customers.

During the course of the project something that we have learned is that just because your project works with the latests stable dependencies, does not mean that there will not come some last minute changes, that will break it. This was especially bad at the last day of sprint 4, making for an annoying experience, when we were “finished”. So a very important thing, is good communication between the groups, to avoid such situations.

8.1 Sprint conclusions

As the project has run in sprints, we will make a small conclusion for each sprint, as well as for the collaborative works, in the following sections.

8.1.1 Sprint 1

Our task for sprint 1 was to rewrite the local database schema, so that it matched the global database schema. We got this task from the planning meeting of sprint 1. We fulfilled the task and rewrote the local database schema so it was ready to store the intended information

from the applications in the GIRAF project. Because of this sprint we got to know the database, which gave us a better starting point in sprint 2.

8.1.2 Sprint 2

In sprint 2, we started our work on the application Cat. We quickly discovered that the primary task of this sprint, would be to get Cat to work with the local database. Due to the knowledge on the database we gained during sprint 1, we only needed to get acquainted with Cat, saving some time. By the end of sprint 2, we had technically fulfilled our goal of getting Cat to work with the new database, as it did not show in the GUI when changes were made, becoming a task for sprint 2.

8.1.3 Sprint 3

The tasks in sprint 3 included getting Cat to show when changes were made. After this was fulfilled and the core functionalities of Cat was completed, we started on some of the functionalities that would make Cat more easy to use, so the users of Cat did not have to perform the same monotone actions over and over again. Another big change that happened to Cat in this sprint, was the GUI of Cat that got updated a lot. Some of the functionalities that was not completed in this sprint, were the focus of sprint 4.

8.1.4 Sprint 4

In this last sprint we wanted to do several things before the project concluded. We wanted to implement some of the most important features left from sprint 3. We also wanted to refactor and improve the code to make it more readable, to ensure that it would be easier for future groups to get acquainted with the code. Furthermore we wanted to ensure that Cat worked as intended, so we made some functionality tests on the features in Cat.

We feel that this sprint was the most productive of the sprints, and completed a lot of what we wanted. We implemented features such as copying and refactored parts of the code quite a lot. We also removed some of the classes, that were no longer used. This made Cat much more readable, and we hope that the students next year will find it more intuitive than we did, when we first received Cat.

Our tests showed that Cat generally behaved as expected, which meant that we achieved what we wanted.

8.1.5 Common writings

Writing in collaboration with other groups has the potential to be a natural part of the project, when working together on a part of the system, as in Chapter 6. Otherwise, it will feel forced, which it did when we did a code review in Chapter 7. That is not to say that it was not useful, because it *was* useful, as we got more tips on improving the code.

8.2 Project experiences

This section contains experiences we made during the project, both with working in a multiproject, and using new tools.

8.2.1 Taking over a project

Working on something others have started was new to both of us, and resulted in a lot of trial-and-error learning.

In sprint 1, when working on LocalDB, this resulted in some time "wasted" trying to figure out if the source structure was good or not, both discussing it, and trying alternatives. In hindsight, it was actually quite good, with a clear distinction of which classes did what.

It was even worse when receiving Cat, both due to a larger project, and a lower quality of the code (which we blame the previous year for, as the methods made by the group working on Cat during sprint 1 was of a higher quality).

8.2.2 Collaborative working

When working in a multi-project, communication is paramount, even more so than when working in only one group, as the location is physically separated (even if only by a few meters and two doors). In sprint 1, we worked together with the two other database groups (sw612f14 and sw615f14), with whom we discussed who should do what, in the beginning of the sprint, and then we each worked in our own groups. Late in the sprint we had some problems between Oasis and LocalDB. These problems could have been found, and worked out, much earlier if we had talked more with each other earlier on.

8.2.3 Version control

We do of course have experience with version control, but have until this project only used SVN. After getting started using git we have come to realize it is more powerful than SVN. Two usages we found especially useful: branches, and the ability to commit locally, without everybody else getting access to the changes. We only used branches minimally, having a "dev" branch while working, and using the master branch as the (more or less) stable release branch. The local commits made it possible to record changes on the fly, improving the ability to undo later, without sharing something incomplete.

Unfortunately, using git was not without trouble, as the groups have not been consistent with what branches have been functional, and thus, some time was spent pulling down the proper versions and installing them. This was especially problematic when working on Cat, which depended on several other projects.

8.2.4 Development on Android

While it was interesting to develop on a new system (neither of the group members have developed for Android before), it was an annoying experience, especially in the beginning. The IDE (Android Studio 0.4.6) is quirky, slow and a bit limited compared to Visual Studio 2012, which we are used to (and prefer). During sprint 1, we also had problems with one installation of Android Studio, as it refused to update, even after reinstalling (the new version was not available as a download). The solution was to manually start the updater, which then worked without problems.

Finally, the default emulator is incredibly slow. This can be solved using Intel HAXM [12], which brings it to near native speed on supported systems. To loosely illustrate how slow the default emulator is, installing the Launcher took more than two minutes, compared to around 10 seconds using HAXM. As a side note, it is important to use the most recent version of HAXM, as earlier versions may give BSOD at random times (but only after having been launched).

8.3 Recommendations for swf15

This section contains our recommendations for sw6f15, on several topics regarding the project.

8.3.1 Coding in a multiproject

Assuming you will be using git (which you really should), and are not already familiar with it, spend a couple of hours reading the first three chapters in the book "Pro Git" [13] (available online for free [14]). And make sure that the very first thing you do when you receive a project is to make a development branch to work on in your group, and only merge into the master branch when stable.

8.3.2 Receiving a project from another group

If you are the first group in the semester to work on a project, make sure to take the time to understand what is actually going on in the code - even if this means you have even shorter time to actually work on it. This is especially important if you will work on the code for more than one sprint.

8.3.3 Developing for Android

If you continue to work with Android, and intend to use the emulator, we strongly recommend you to continue in Android Studio, and use the HAXM [12] emulator. If nothing else, because the groundwork for doing so have been made this year.

8.3.4 Report writing

Assuming you work in several sprints, make sure you document each sprint while working on it, or at the very latest, at the beginning of the next sprint. It is also a good idea to save the commits for the start, and end, of each sprint, so you can easily verify that something you have documented in your report have not been changed in the code.

8.3.5 Communication in a multiproject

If you do weekly status meetings like we did, make sure to keep them short, and take discussions later with the related groups.

8.4 Future work

The following tasks where the ones that were not completed in 2014, but could be made in the future:

- Ensure only Giraf-Components are used.
- Global categories - we have only worked with categories associated with a single citizen so far.
- Copy a subcategory to another category.
- Pending icon when not synced.
- Sync button.
- Play sound associated with pictogram.
- Possibly remove CategoryLib - it is just a wrapper for Oasis.

Ensure only Giraf-Components are used The class `SettingsDialogFragment` still does not use Giraf-Components for the GUI everywhere, therefore this should be fixed in the future, to ensure that Cat and GIRAF are consistent in their GUI.

Global categories - we have only worked with categories associated with a single citizen so far Global categories would make it easier for the guardians to choose certain categories, for a specific citizen, and thereby setup categories, subcategories and pictograms for the person.

Copy a subcategory to another category This functionality is similar to copying a category to another profile, although it is not quite as important. But it would still be a feature that would make Cat more usable.

Pending icon when not synced When changes have been made, there should be a pending icon in all applications, if the changes has not been added to the global database, to ensure that the guardians is aware of this.

Sync button It should be possible to manually synchronize with the global database from all applications in GIRAF including Cat.

Play sound associated with pictogram All applications should be able to play the sound associated with a pictogram.

Possibly remove CategoryLib - it is just a wrapper for Oasis. A last task would be to remove CategoryLib completely. We have done so in Cat's `MainActivity`, but some of the other classes in Cat are still dependent on it. This could be done by accessing Oasis directly.

Chapter 9

Bibliography

[1] A. Vinther, C. K. Hansen, J. E. M. Hilker, and L. Nielsen, "Communication application for children with autism, category administration and inter-application communication," student report, Aalborg University, June 2013. 11, 13, 60

[2] B. Flindt, H. L. Magnussen, J. B. Tarp, and S. Jensen, "Wasteland - a giraf database," student report, Aalborg University, June 2013. 14, 42

[3] A. Developers, "Storage options." http://developer.android.com/guide/topics/data/data-storage.html#db. 44

[4] L. Vogel, "Android sqlite database and content provider - tutorial." http://www.vogella.com/tutorials/AndroidSQLite/article.html. 44

[5] SQLite, "About sqlite." http://www.sqlite.org/about.html. 44

[6] "Sql-92." 44

[7] SQLite, "Sql features that sqlite does not implement." https://sqlite.org/omitted.html. 44

[8] A. Developers, "Content providers." http://developer.android.com/guide/topics/providers/content-providers.html. 44

[9] "Code review." http://en.wikipedia.org/wiki/Code_review. 45

[10] J. Cohen, "11 proven practices for more effective, efficient peer code review." http://www.ibm.com/developerworks/rational/library/11-proven-practices-for-peer-review. 45

[11] "How do you perform code reviews?." http://stackoverflow.com/questions/310813/how-do-you-perform-code-reviews. 45

[12] H. J. (Intel), "Intel®hardware accelerated execution manager." https://software.intel.com/en-us/android/articles/intel-hardware-accelerated-execution-manager. 51, 52

[13] S. Chacon, *Pro Git*. Apress, 2009. 52

[14] S. Chacon, "Pro git book." http://git-scm.com/book/en. 52

[15] A.D.A.M., "Autism." http://www.ncbi.nlm.nih.gov/pubmedhealth/PMH0002494. 60

[16] C. Nordqvist, "What is autism? what causes autism?," *Medical News Today*, 2013. http://www.medicalnewstoday.com/info/autism. 60

Part I

Appendix

Appendix A

Specification requirements

This chapter contains the requirements for LocalDB and Cat. Apart from the numbering of the headers, they are a verbatim copy of the requirements specification made by groups sw601f14 and sw604f14, during sprint 1.

A.1 Requirement Prioritization

The requirements in this section is split into the different application they belong to. Each individual requirement has a number in end, which shows the priority of this specific requirement.
(1) - Requirements that has the number (1) is a requirements that must be fulfilled.
(2) - Requirements that has the number (2) is a requirements that must be fulfilled, but which are not essential.
(3) - Requirements that has the number (3) is a requirements that must be fulfilled, but only if there is additional time available.

A.2 General

- No applications must close, stop or crash unexpectedly. (1)
- All applications must synchronize with the global database, once a day, if there is access to the Internet. (1)
  - It must be possible to synchronize manually with the global database. (2)
- All applications must be able to save settings for each individual profiles. (1)
  - It must be possible to copy settings from one profile to another. (2)
- All applications must run in landscape mode (horizontal). (1)
  - Sequences and Week Schedule must be able to use portrait mode. (1)
- All applications, except the Launcher, must have a 'close' button. (2)
  - When the button is pressed, it must close the application, if the user has permission to close applications. (1)

  - If the button is pressed and the user does not have access to close the application, the application must be able to close with the help of a guardian, e.g. by scanning QR code. (1)
- When and only when there is a permitted input, there must be some kind of response, which makes sense in context, but otherwise this is up to the individual developer. (2) see user story number 4.
  - Applications with functions targeted towards citizens must only use singleclick, drag and drop, and swipe input. (2)
  - If there is a function, which should only be usable by a guardian, then all Android gestures are permitted. (2)
  - All applications must be named with meaningful Danish names. (1)
- If a pictogram is not yet saved to the global database, then all applications must show a 'pending' icon, so the guardian can see that it is not yet saved to the database. (2) see user story number 8.
  - Pictograms must be able to be marked as private, so they are not uploaded to the global database. (2)
- All applications must be able to play a pictogram's associated sound. (2)

A.3 Database

- Pictograms, photos, sound clips, profiles and categories must be able to be saved in the database. (1)
  - Citizen profiles are split into groups. (1)
  - Profiles must be split into organizations. (1)
  - Pictograms and photos must be handled the same way in the database. (1)
  - Sound clips must be handled separately. (1)
  - Categories must be handled separately. (1)
- Global database:
  - It must be possible to synchronize the local database with the global database. (1)
  - Must contain all standard pictograms. (2)
  - Guardian profiles must be able to add new pictograms to the database. (1)
  - Guardian profiles must be able to remove and edit pictograms they have added to the database. (2)
  - Guardian profiles must be able to copy standard pictorgrams, edit them and then save the copy to the database. (2)
  - Guardian profiles must not be able to remove standard pictograms. (3)
- Local database:
  - All contents of the local database must be able to be saved in the global database, this includes settings. (1)
  - It must be possible to synchronize with the global database. (1)
    - It must be possible to synchronize manually. (1)
      - There must be a synchronize button for the guardian in all applications. (2)
    - It must be possible for the synchronization with the global database to happen automatically. (2)
      - As standard, automatic synchronization should happen once a day when there is wifi. (2)
  - All applications must use the same local database, e.g. different application does not have their own local database. (2)

A.4 Categorizer

- Must be able to create new categories of pictograms. (1)
- Must be able to view, add and remove pictograms from existing categories. (1)
- Must be able to delete existing categories. (2)
  - When a category is deleted, a warning dialog box must be shown asking the guardian if they are sure they want the category to be deleted. (2)
- Guardian profiles must be able to connect a specific category to a specific citizen profile. (1)
  - It must be possible to copy a category from one profile to another. (2)

Appendix B

Keywords

Person with autism

Autism is a group of disorders affecting brain development. People with autism usually have difficulties in social interaction and communication, both verbal and non-verbal. The disorders are often observable within the first three years of a childs life.

Autism can not be cured, but an early treatment program can greatly improve the life quality of a person with autism. Most treatment programs are based on the interests of the person with autism through a highly structured schedule of activities that benefits the person with autism [15, 16].

Guardian

A guardian is a caretaker of the people with autism. They have administrative privileges in the GIRAF project, and are for example allowed to create categories and subcategories, and create the relations between them and relevant pictograms in the application Cat.

Profile

In GIRAF guardians and people with autism are represented by profiles. Profiles can have different associations to for instance categories, subcategories, pictograms and other profiles depending on whether it is a guardian or a person with autism.

Pictogram

A pictogram is an image representing a living being, a physical object or some form of action [1, p. 25]. The main purpose of a pictogram is to convey the meaning of the resemblance between a given object or action and an image. To enhance the comprehensibility of a pictogram, a text label can be added describing it. Pictograms are used as a mean of communication and a way of expressing certain things. They are commonly used by people who need assistance with communication, as for instance some people with autism.

Category

A category is a collection of subcategories and pictograms. In Cat the relations between a category and its subcategories and pictograms can be created. The purpose of categories is to help people with autism in their daily life, by giving them some guidelines to follow. An example of a category for a person with autism could be food. This category could contain pictograms of food and subcategories such as breakfast, lunch and dinner. These subcategories could also contain relevant pictograms. The categories and their relations can be created by guardians.

Appendix C

Database schemata

There is not enough space on this page for the figures, so please go to the next page.

C.1 Database schema after sprint 1

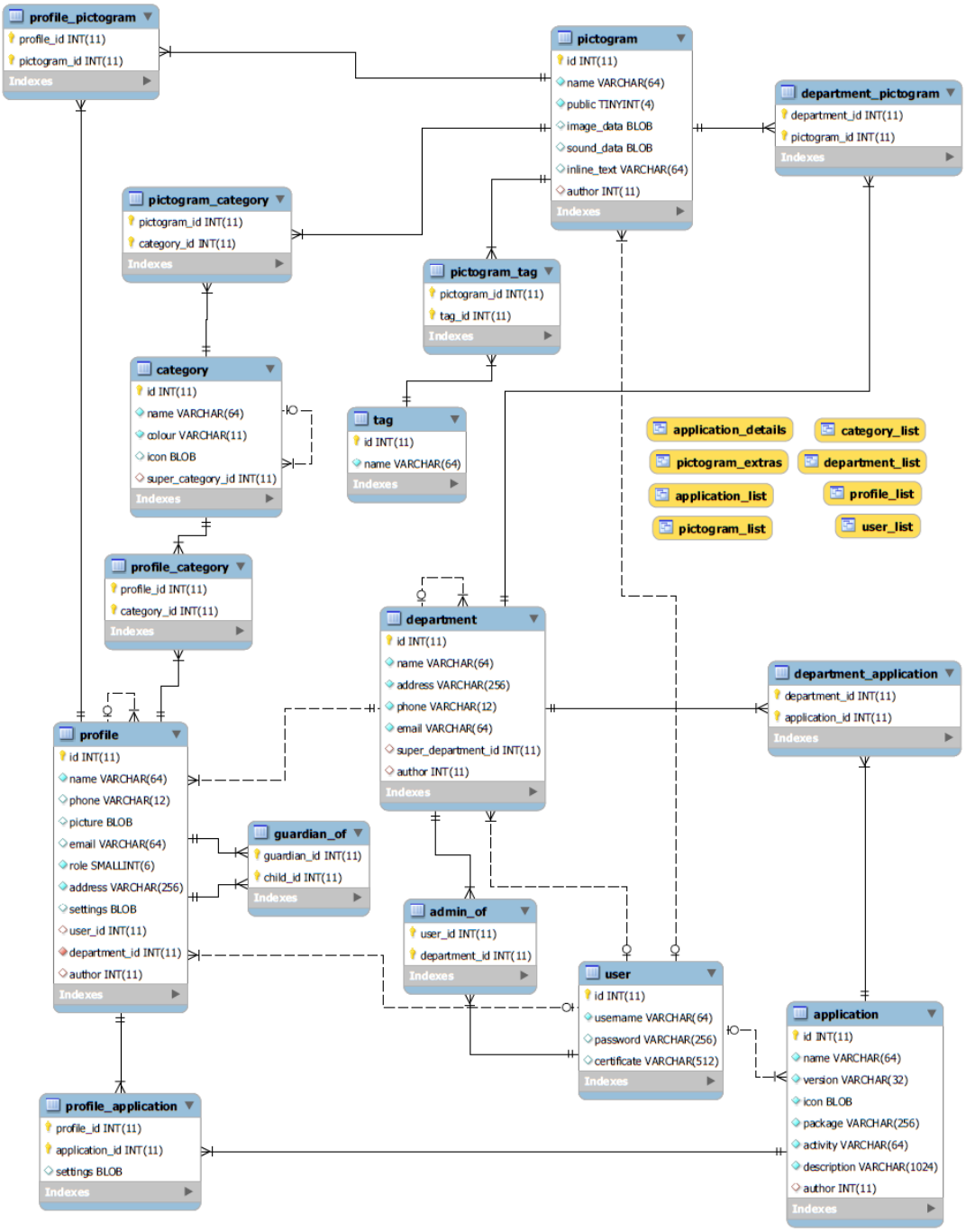

Figure C.1: The database schema implemented in Local DB in sprint 1.

C.2 Database schema before sprint 1

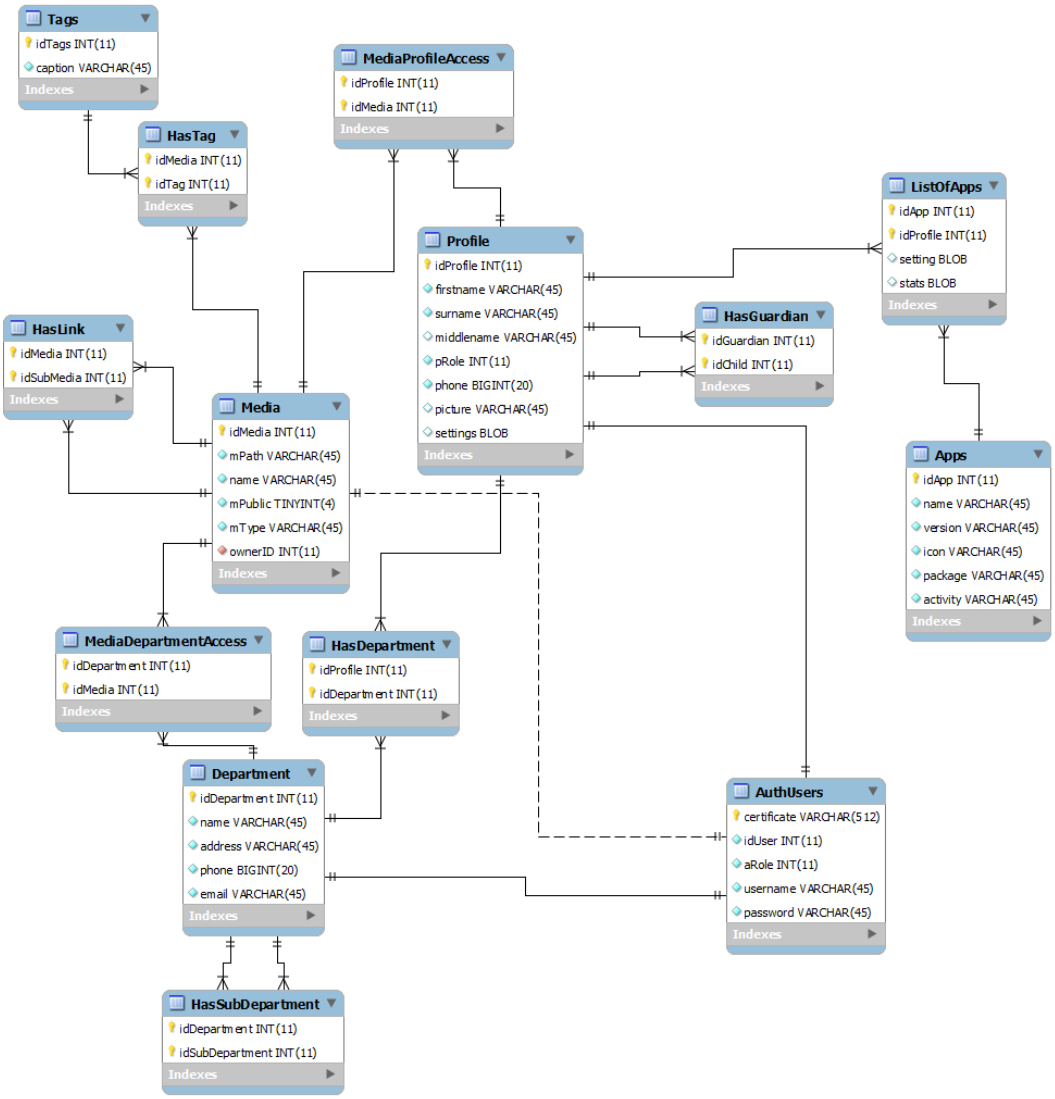

Figure C.2: The previous database schema.

Appendix D

Test results

This chapter contains our functionality tests, with test cases, expected results, and actual results.

Create category

Create a category with no titel, no color, no icon.
Expected: Error.
Result: Expected.

Create a category with titel, no color, no icon.
Expected: Error.
Result: Expected.

Create a category with no titel, color, no icon.
Expected: Error.
Result: Expected.

Create a category with no titel, no color, icon.
Expected: Error.
Result: Expected.

Create a category with titel, color, no icon.
Expected: New category with title and color, but no icon.
Result: Expected.

Create a category with titel, no color, icon.
Expected: Error.
Result: Expected.

Create a category with no title, color, icon.
Expected: Error.
Result: Expected.

Create a category with titel, color, icon.
Expected: New category with title, color and icon.
Result: Expected.

Other tests regarding category

Create a category, change profile, change back to the previous profile.
Expected: Category is still in the first profile.
Result: Expected.

Delete a category.
Expected: Category is removed entirely.
Result: Expected.

Close

Close application by removing it from the tasklist, open it again.
Expected: The profiles is as before with their categories.
Result: Expected.

Close application by close button, open it again.
Expected: Choose profile opens.
Result: Expected.

Close application by back button, open it again.
Expected: Choose profile opens.
Result: Expected.

Profile

Choose citizens profile.
Expected: Profile is selected.
Result: Expected.

Change citizens profile.
Expected: New profile is selected.
Result: Expected.

Chose active profile.
Expected: No changes.
Result: Expected.

Choose citizens profile, change citizens profile, select guardian from profile list.
Expected: Error and no changes.
Result: Expected.

Do not choose a profile.
Expected: Error.
Result: Expected.

Subcategories

Add subcategory with title, color and icon to category.
Expected: Subcategory with title, color and icon is added to category.
Result: Expected.

Delete subcategory.
Expected: Subcategory is removed entirely.
Result: Expected.

Pictograms

Add a pictogram to a category.
Expected: The pictogram is added to the category.
Result: Expected.

Add multiple pictograms to a category
Expected: The pictograms are added to the category.
Result: Expected.

Add a pictogram to a subcategory.
Expected: The pictogram is added to the subcategory.
Result: Expected

Add multiple pictograms to asubcategory.
Expected: The pictograms are added to the subcategory.
Result: Expected.

Delete a pictogram from a category.
Expected: pictogram is removed from the category.
Result: Expected.

Delete a pictogram from a subcategory.
Expected: the pictogram is removed from the subcategory.
Result: Expected.

Settings

Category

Long press on a category.
Expected: Edit dialog opens.
Result: Expected.

Press cancel on the edit dialog for category.
Expected: Edit dialog closes.
Result: Expected.

Edit dialog, change title on a category.
Expected: The title on the category has been updated to the new title.
Result: Expected.

Edit dialog, change title on category, with empty title.
Expected: Error.
Result: Category now has an empty title.

Edit dialog, change color on a category.
Expected: The category now has new color.
Result: Expected.

Edit dialog, change icon on a category.
Expected: The category now has the selected icon.
Result: Expected.

Edit dialog, change icon on category, choosing no icon.
Expected: Category now does not have an icon.
Result: Expected.

Edit dialog, copy category to the active user.
Expected: Error.
Result: Expected.

Edit dialog, copy category to another profile with no category with the same name.
Expected: Category is copied.
Result: Expected.

Edit dialog, copy category to another profile with a category with the same name.
Expected: Error.
Result: Expected.

Subcategory

Long press on a subcategory.
Expected: Edit dialog opens.
Result: Expected.

Press cancel on the edit dialog for a subcategory.
Expected: Edit dialog is closed.
Result: Expected.

Edit dialog, change title on a subcategory.
Expected: The title on the subcategory has been updated to the new title.
Result: Expected.

Edit dialog, change titel on a subcategory, with empty title.
Expected: Error.
Result: Subcategory now has an empty title.

Edit dialog, change color on a subcategory.
Expected: Subcategory now has new color.
Result: Expected.

Edit dialog, change icon on a subcategory.
Expected: Subcategory now has the selected icon.
Result: Expected.

Edit dialog, change icon on subcategory, choosing no icon.
Expected: Category now does not have an icon.
Result: Expected.

Edit dialog, select copy for subcategory.
Expected: Warning about not possible.
Result: Expected.

Select

Selecting an item, shows that it is selected by an animation.
Expected: Item shows animation.
Result: Expected.