REVIEW MENSA UNIBE
TEAM 5

Written by Team 2

Autumn semester 2013

submitted to Mircea F. Lungu

Andrea Caracciolo, Andrei Chiș, Bledar Aga

Submitted by

Thomas Steinmann 11-918-331
thomas.steinmann@students.unibe.ch

Karan Sethi 11-924-826
karan.d.sethi@students.unibe.ch

Lea Wollensack 11-114-006
lea.wollensack@students.unibe.ch

Michael Scheurer 11-107-604
michael.scheurer@students.unibe.ch

1. Table of Contents

2. Introduction

We are writing this review regarding the “Mensa App Project” “Team 5” created. We are looking forward to look at the project and hope that we will be able to help this team.

It is a pity that the application crashes just after the main activity is executed. Therefore we will not be able to say much about the design. In additon the link of the fluidui does not work. We would appreciate it if in the near future these things would work.

3. Design

Violation of MVC layers

The MVC pattern is used in the project. Some slight violations of the MVC layers could have been found though. To detect those we would like to go through the 3 layers Model, View and Controller step by step.
The Model layer consists of the data, the logic and all the functions. In the mensaunibe project the reviewer broadly suggests the following packages to be part of the Model: *com.ese2013.mensaunibe.model, com.ese2013.mensaunibe.util, com.ese2013.mensaunibe.util.database* and *com.ese2013.mensaunibe.util.database.table*. These fetch the data with JSON, handle it and store it in the database. The Model.java class for example also checks whether a redownload of the data is needed. From our understanding of the Model layer, all the classes responsibilities are within this layer's responsibility.
The View layer is the output representation of the data. This is found in the layout. The com.ese2013.mensaunibe package is seen as part of the View layer. Here, a violation is found in the ActivityMain.java class. This class does not only define the app's behaviour in direct user interaction but also takes up responsibilites of the Controller layer and even the Model layer. For example, it fetches the data from the API with the help of JSON. This is clearly a task which should be executed by a class from the Model layer. Moreover, the class also checks whether the user is already registered in the API. This should be a function in a class belonging to the Controller layer. We will go into details about the ActivityMain.java in the chapter "ActivityMain".
The Controller layer is responsible for the update of the model's state. As part of this layer we can see the *com.ese2013.mensaunibe.util.gui* package. The Adapter classes in this package extend the Adapter classes provided by Android. On the Android Developers site the Adapter object is described as "a bridge between an AdapterView and the underlying data for that view". So it takes up precisely the functions of a Controller.

Usage of helper objects between view and model

As described above, all the classes which belong to the Controller layer are some kind of helper objects. The AdapterCustomMensalist object for example helps to handle - amongst other things - the app's behaviour if the user clicks on the Favorite button of a Mensa. A good feature here is that there are toasts sent to the user whenever the app navigates to a new page. This responsiveness helps to increase the app's usability.
Also in the Model layer itself we can find helper objects, for example the MensaDatabaseHelper. The code makes use of the Google Play Services Library too.
Generally, the code makes good use of helper objects between the view and the model.

Rich OO domain model

To evaluate the richness of the project's domain model, an overview of all the classes e.g. an UML Diagram or something similar would have been of great help.
In general, the domain model is object-oriented, although not thoroughly. Looking at the Navigation Prototype we expect an object for every part of it. If we compare the Navigation

Prototype with the Android Application project we can see that the objects Friends, MensaDetails, Mensalist, Map, Menulist, Notifications and Settings are implemented as Fragments in the code which is good. But for example the object Favourite Food cannot be found as class name in the project, indicating that there is no such object. We then find the feature of favorising a Mensa in the AdapterCustomMensalist. An issue here is that some classes have unclear responsibilities which we will evaluate shortly in the next section.

Clear responsibilities

The responsibilities of the packages are clear.
Nevertheless, there are classes which do not have a clear or more than one responsibility. An example of this is the ActivityMain class which will be reviewed in more detail in the analysis of an activity. If we follow the Single-Responsibility-Principle this class should be refactored. As another example it is questionable if the AdapterCustomMensalist object should also get fewer responsibilities; now it helps in showing the Mensadetails for a clicked Mensa, directs the User to the map if he clicks on the navigation button and also sets a Mensa as Favourite. At the other hand the object tries to help handling everything which concerns the Mensalist view. The responsibility-driven design of this app could be improved.

Sound invariants

Unfortunately, no invariants could be found in the code. Nevertheless it would in some cases be helpful to have one. For example in the Model.java class a method could check if there is an internet connection before the method isRedownloadNeeded() is executed. This could be done using the isConnected() method provided by the Android API.

Overall code organization & reuse

Honestly, the code organization could be improved. Although the structurization into the different packages is quite well done, it was not very easy for the reader to understand the code due to its organization. This comes from the unclear responsibilities of some classes. Regrettably, no interfaces are used, neither for the view nor anywhere else. On the other hand, classes provided by Android are inherited and its methods used.

Conclusion

The app's code design is on a good way but can definitely be improved. Especially the team should have an eye on the Single-Responsibility-Principle. In order to achieve a better design graphical methods like UML Diagrams or CRC cards could help.

4. Coding Style

Introduction

In this section we will discuss the general style of the code written in the Mensa Unibe project. Evaluation criteria will be as follows, according to the lecture Introduction To Software Development.

Consistency

The Code is consistent in general, except in the FragmentMenuListDay and FragmentMenuListAllDay classes which can be instantiated via a static newInstance method, while no other Fragments have this option. This means that different fragments in the same project have to be treated differently which is inconvenient for the programmer using the fragment.

Method Names and Comments

Most methods are named properly, demonstrating what they do accurately. However there are many redundant comments describing the exact same thing the method name already describes. These comments only fill up whitespace that would be useful for readability and tire the eye of the reader.
On the other hand, it would be great if some javadoc would specify the contract of the various methods, since it can also be accessed inside the code where a method is called.

Helper methods and reusing code

In general there could be a lot more private methods to split up big walls of code into small, readable blocks. This applies most apparently in the ActivityMain classes onCreate Method which contains several big if statements as well as multiple submethod definitions all in one method. This huge method could be split up very easily with the help of eclipse's refactor functionality to group the initialization of various elements like the WebServiceAsync.

Possible refactoring:

```
private void setupWebService(Boolean registered) {
        if ( !registered ) {
                new WebServiceAsync(new AsyncCallback() {
                        @Override
                        public void onTaskDone(JsonObject json) {
                        //…
                        } else {
                        //…
                        }
                }
        }).execute( /*url here*/)

        } else {
                Toast.makeText( */context*/)
        }
}
```

The same principle applies also to the onDraw method in the TitlePageIndicator class and several other methods, though these two are the most alarming.

Exceptions and null values

There are three distinct custom Exceptions that are thrown in the event they are designed for. They are used when the device is not connected to the internet or the database fails to handle a request. The Exception handling is done very clean overall. However in getOverflowMenu, Exceptions are caught generally by catch {Exception e}, which nullifies the point of creating different exceptions.

Assertions, Contracts and Invariants

There are no assertions or invariant checks in the code, which means no responsibilities of a class are tested and neither are any method contracts. These would ensure that classes did not violate their responsibilities and methods uphold their contracts, making it a lot easier to determine where a mistake happened in the debuging process.

Encapsulation

The ActivityMain class contains a lot of public variables which violates encapsulation. The comments suggest they need to be public so that they can be used by the fragments embedded into the activity, but these could still use the abilities set- and get methods , if they were implemented. Alternatively primitive data can also be sent to the fragments via an intent.

5. Documentation

Understandability

The whole documentation is easy to understand. Especially the Use Cases are written in a clear language and have a good structure.
Only in Use Case #12, Menu's rating, something is not quite understood: how is it possible for the user to rate a menu he/she has already eaten by going to the "Upcoming menus" tab? Should not there be future menus listed?

Intention-revealing

The introduction makes the app's features very clear. Also the Use Cases reveal details of these features. Nevertheless the real intention behind the whole app, its purpose, is not mentioned. The reader can only guess that the app's intention is to make the mensas more popular and more attractive.

Responsibilities

The app's responsibilites are clear. The UI Prototype illustrates the six tabs: Home, Mensa List, Mensa Menus, Mensa Map, Notifications andFriends. Also the Use Cases demonstrate those tab's responsibilities, although there is an inconsistency in the naming as explained below.

Consistent domain vocabulary

In general the vocabulary is used consistently. Nevertheless there is an inconsistency between the Use Cases and the UI Prototype as far as the reader understood them correctly.
In Use Case #1, #2, #5, #6, #11 and #12 there is a clickable item "Mensas" mentioned. If the reader compares this with the UI Prototype there is no clickable item "Mensas" found. The reader is guessing that this "Mensa" refers to the tab "Mensa List".
Also in Use Case #3 there is a clickable item "Upcoming Menus" which cannot be found whether on the UI Prototype nor on the Navigation Diagram.
The same inconsistency is found in Use Case #4: As a second action the user clicks on "Closest Mensa" in the menu. Checking the UI Prototype there is no "Closest Menu" tab found, only one named "Mensa Map" which is thought to have the same functionality as the "Closest Mensa" tab.
When looking at the UI Prototype it is not clear where the section "Settings" mentioned in Use Case #13 and #14 would be. It was not found on the Homescreen.

All in all

The documentation is well done and clearly structured. The naming inconsistency between the Use Cases, the Navigation Diagram and the UI Prototype could be optimised. It is a pity that the reviewer could not have a look at the app in action or even the Fluidui Prototype. There always appeared the following error message when opening the link: “Couldn’t download correctly (has it been deleted/made inactive?)”.
(https://www.fluidui.com/editor/live/preview/p_vXiOFtiAveEFxEmmFLqaOSMrlEGhKvUE.1381322284083)
Also the XML Document “Use Case Diagram” could not be opened in a readable way.

6. Testing

Before we start with the review for the tests, it is necessary to mention that there is only one testcase class defined, with no tests implemented.

Clear and distinct test cases

We cannot say anything about clearness and the test distinction, because of the reason mentioned above.

Number/coverage of test cases

It is very difficult to talk about the coverage of the test cases, when there only exists one test case, where no tests are implemented.

Easy to understand the case that is tested

We cannot review this point, because of the lack of tests.

Well crafted set of test data

It is very difficult to say anything about the craftiness of the test data, therefore we will be giving no comment to this.

Readability

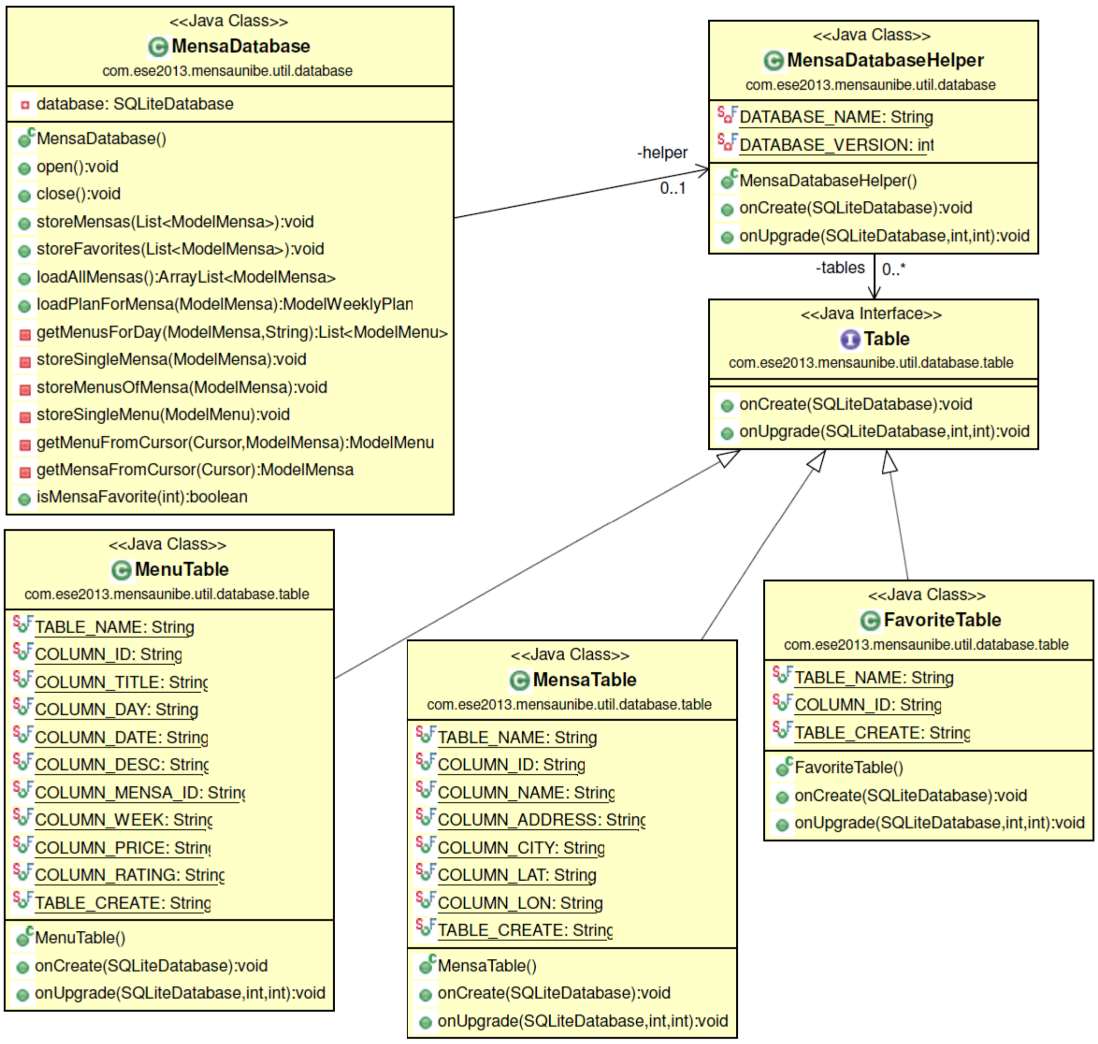
The readability of the test case that has been created is fine. The case is well structured and has all the methods a test case must have. But it is still important that some tests actually run.

Conclusion

To conclude we would just like to mention that for the next time, please include tests for this project. It will help you find bugs and write the code of your project much easier. It can even help if the tests are implemented before starting with the code of the project. This will make things easier and you will know that everything works if none of the tests are failing.

7. Data-storage

To get an overview of how the data are stored, it's usefull to have a look on the UML-Diagramm:


<<Java Class>>
MensaDatabase
com.ese2013.mensaunibe.util.database
database: SQLiteDatabase
MensaDatabase()
open():void
close():void
storeMensas(List<ModelMensa>):void
storeFavorites(List<ModelMensa>):void
loadAllMensas():ArrayList<ModelMensa>
loadPlanForMensa(ModelMensa):ModelWeeklyPlan
getMenusForDay(ModelMensa,String):List<ModelMenu>
storeSingleMensa(ModelMensa):void
storeMenusOfMensa(ModelMensa):void
storeSingleMenu(ModelMenu):void
getMenuFromCursor(Cursor,ModelMensa):ModelMenu
getMensaFromCursor(Cursor):ModelMensa
isMensaFavorite(int):boolean
-helper
0..1
<<Java Class>>
MensaDatabaseHelper
com.ese2013.mensaunibe.util.database
DATABASE_NAME: String
DATABASE_VERSION: int
MensaDatabaseHelper()
onCreate(SQLiteDatabase):void
onUpgrade(SQLiteDatabase,int,int):void
-tables
0..*
<<Java Interface>>
Table
com.ese2013.mensaunibe.util.database.table
onCreate(SQLiteDatabase):void
onUpgrade(SQLiteDatabase,int,int):void
<<Java Class>>
MenuTable
com.ese2013.mensaunibe.util.database.table
TABLE_NAME: String
COLUMN_ID: String
COLUMN_TITLE: String
COLUMN_DAY: String
COLUMN_DATE: String
COLUMN_DESC: String
COLUMN_MENSA_ID: String
COLUMN_WEEK: String
COLUMN_PRICE: String
COLUMN_RATING: String
TABLE_CREATE: String
MenuTable()
onCreate(SQLiteDatabase):void
onUpgrade(SQLiteDatabase,int,int):void
<<Java Class>>
MensaTable
com.ese2013.mensaunibe.util.database.table
TABLE_NAME: String
COLUMN_ID: String
COLUMN_NAME: String
COLUMN_ADDRESS: String
COLUMN_CITY: String
COLUMN_LAT: String
COLUMN_LON: String
TABLE_CREATE: String
MensaTable()
onCreate(SQLiteDatabase):void
onUpgrade(SQLiteDatabase,int,int):void
<<Java Class>>
FavoriteTable
com.ese2013.mensaunibe.util.database.table
TABLE_NAME: String
COLUMN_ID: String
TABLE_CREATE: String
FavoriteTable()
onCreate(SQLiteDatabase):void
onUpgrade(SQLiteDatabase,int,int):void

SQLite-Database is used in the app to store the local data . The self-written MensaDatabaseHelper-Class extends the android SQLiteOpenHelper-Class and overrides the methodes onCreate() and onUpgrade(). This Helper-Class is responsible for upgrading Database-Versions and all CRUD-operations (Create, Read, Update and Delete).
The tables are implemented in a way that the interface Table requires two methodes onCreate() and onUpgrade(). It's not sure if the interface is really required. In our opinion it generates more code than necessary. The tables could simply be created in the onCreate() method inside the Helper-Class. Though the SQL-statements in the three table-classes (Mensa, Menue, Favorite) are consistent and use the usual and proper statements like "primary key", "not null" and "type". In additon unique id's are used in each table. Only in the favorite-table we can find only one column named _id. It is not clear for which related table the favorite-table is designed for. We propose to use a designation here, like "mensa_id".

As a last Class, the Mensa-Database provides a lot of usefull methodes to store and read data. This part of the code is proper, consistent and uses clear method-names. Maybe the open() method could be called in each method. Because it can be forgotten to open and close each time the database is used.

8. Activity of choice

In the project we will be talking about the main activity class. It is the only found activity in this project:

The main activity handles all the fragments and also binds them. Everything in this application works through this activity. It is the only activity defined in this project.

Our first impression by looking at the code is, that there are very little methods and some methods have too much code. That makes it very difficult to go through the activity and understand what it is actually supposed to do. There is also a lot of code that has not been deleted, but commented out. All these points make this activity very unstructured and complicated at the first look.

When we go through the code, we can see that there are a lot of global variables. For encapsulation reasons we would like to suggest all of them to be private and accessible via additional getter methods.

When we continue reading the code, we can see that this activity also gets the data from the API with the help of json. The main activity already has the responsibility to handle all the fragments. Therefore it would be better to make a separate class to collect data from the API, for example a “JsonClass”, if we want to follow the rule of responsibility driven design.

Moreover in the next lines, there is a “locationClient” defined which tries to connect with the google location service. This code should be moved into a different class. The reason is the same as above. It is important, that every responsibility, that can be given to another class should be delegated.

When we continue reading the code, we can see a method “calcWeekDay()”, which should be implemented in a different class, because the main activity should be responsible for the fragments and not for the calculating something about the weekday. The solution would be that a new class can be created, for example “WeekDay” or “Date” which can be responsible for calculating a coming specific weekday. Perhaps in near future, there could be a possibility to expand the project and therefore it would be nice to have a class Date. Also this class could be reused in other classes where the weekday is calculated seperately.

To conclude we would just like to say, that the code could have been better structured, using shorter methods, so that readers get an idea what each class does. It was anyhow very difficult for us to review this activity because we were not able to get the application running.

9. Conclusion

To conclude we want to state that there is still a long way to go and there could definitely be some improvements regarding code style and organisation. The way the classes are set up on the other hand, is satisfying and the database looks clean and organized. Sadly we could not see the product in action but there has definitely been a lot of work put into this project.