Cross Review Team 6

Mensa-app

Authors: Nico Färber, Lukas Riesen, Nils Schnabel, Viviane Tanner, Adrian Wälchli

Note: This document is thought to be a feedback for team 6.
It is written by the team 3 members

Outline

1. Review

1.1 First impression of app in action

+ Slick Design: pleasant colored, no fancy and unnecessary animations
+ clearly structured, not overfilled with information
+ intuitive
+ A nice logo
o Needs internet connection on first use (according to SRS Precondition).
o Probably not quite finished.
- Unstable: Crashes when back button is pressed in map-view.
- It's taking a moment to get used to the interference between the slide-in-menu and the way to slide over for a menu change.

Legend:
+ positive
o neutral
- negative

1.2 Design

Violation of MVC layers

In android there is a different way of implementing the MVC layers. Activities are generally Controller and View. However, in this app it is implemented differently, namly by using Fragments. The Model is clearly distinct from the GUI and we could not find any violations. Fragments and Activity both function as Controller and Views. There is no distinction between these two layers. Additionally, there are custom listeners which also act as Controllers, for example the FavoriteButtonListener.

Usage of helper objects between View and Model

The observer pattern is used to bind the Model and the GUI, so there is no need for helper objects.

Rich OO domain model

Your domain model is relatively small but it looks very solid and well designed. You have distributed the responsibilities to the right objects and you have no unnecessary objects. Encapsulation has been applied, there are getter and setter methods and instance variables are private.

Clear responsibilities

In general, the responsibilities are very well distributed. However, there is missing Javadoc which has to describe these responsibilities.

Sound invariants

We could not find any invariants in the code. At the moment they are not really necessary, but we would recommend to add some for further development. For instance you could look at the day class and say: The month must be between 1 and 12 and the day between 1 and 31.

Overall code organization & reuse, e.g. Views

Generally the code is well organized. However, we had some trouble to understand the Day class first. The Day is basically a Date, it also wraps the Java.util.Date class. Of course, in the context of your mensa app it makes sense to call it a Day.

Overall, the code organization of the code is quite good.

1.3 Coding style

Consistency

The documentation is inconsistent. Some classes have a lot of Javadoc and some have non. The same problem exists for the test-classes and regular comments.

Intention-revealing names

In this app the names for methods and variables are mostly well chosen. In the cases where they are not, Javadoc helps to understand.

Do not repeat yourself

At some places redundant code exists. One example can be found in the Model class, where the methods “mensaLoaded()” and “noMensaLoaded()” both return exactly the same information. On the other hand, there are some classes which inherit from another class with methods that read:

```
someMethod(){
        super.someMethod();
}.
```

This does not make sense since Java does this job anyways (eg. MensaListFragment, DailyPlanFragment)

Exception, testing null values

The app has a wide need for good Exception-Handling. In the most cases it is handled well, but there are cases where exceptions will be thrown through multiple methods and classes. There should be a better use of pre- and postconditions. For example in the Day-class, where the constructor uses a Date parameter, which must not be null respects to the Javadoc. However, there is no assertion testing it.

Encapsulation

Overall there is a good encapsulation. All instance variables except for the static ones are private. Most methods are not public. However, there exists some methods thought to be helper-methods, which are only called within the class, and which are declared as public. A private declaration would be a better choice. As example there is the DailyPlanFragment:

```
public void setDay(Day day) {
        this.day = day;
}
```

May be it will be used in future to be called from the outside. In these case the public declaration would be correct.

Assertion, contracts, invariant checks

We know that android does not support assertions by default, but writing them is good practice anyways. You should use assertions for pre- and postconditions and implement invariants where they are relevant. It is good that you have written the pre- and postconditions in the Javadoc.

Utility methods

In your code you use a lot of helper methods. That is a very good way to make the code more readable and reusable.

Coding conventions/ best practice

Some boolean methods do not follow the Java naming-convention. It states that boolean methods should start with is/has/can… (eg. in class Model).
Sometimes braces are omitted in if, if/else and for statements when there is only one statement inside, which is against best practice (it makes the code harder to read and to maintain).

1.4 Documentation

Understandable

SRS: The SRS contains all relevant informations and is easy to understand.
Javadoc: Generally good and easy to understand (when existing) but there are examples in which the description of the returnvalue is a bit harder to understand, such as in the Day-Class:

```
/**
* Converts the Day to a Java.util.Date. Used internally to allow for
* formatting with the SimpleDateFormat class.
*
* @return Date set the the day represented by this Day object.
*/
```

Intention-revealing

SRS: good
Javadoc: Mostly good, but there are missing a few descriptions for methods. Its understandable, that its not necessary to describe every method. But there are methods which haven't Javadoc yet and are not trivial. Additionally it would be nice to write Javadoc for overridden methods and tests too.
There are missing class descriptions too for example for the DailyPlanFragment, HomeFragment, ...

Describe responsibilities

Its partly hard to see the responsibilities. It would be nice if you wrote a short piece of Javadoc at the top of your class which describes the responsibility of it.

Match a consistent domain vocabulary

We think that your vocabulary is quite good and stays consistent. The names of your classes, methods and variables are chosen meaningful and the code ‘speaks’.

1.5 Test

Clear and distinct test cases

Its very good and clear understandable done by implementing the test cases in different test classes. It is good that you used the setUp and tearDown methods for your test, like in the MensaDataSourceTest.

Number/coverage of test cases

There are a few tests but its not a good coverage. Every class has to be tested, except the exception classes. Additional there are missing test cases for methods in the test classes.

Easy to understand the case that is tested

The tests are small and do what they are supposed to do.

Well crafted set of test data

There is a good set of tested data, but there are missing tests for the contracts. Make sure you also test the borders of the contract.

Readability

Overall, the code is well structured. The names of the tests are chosen correctly and tell you what they test.

1.6 Other

The Back-Button issue

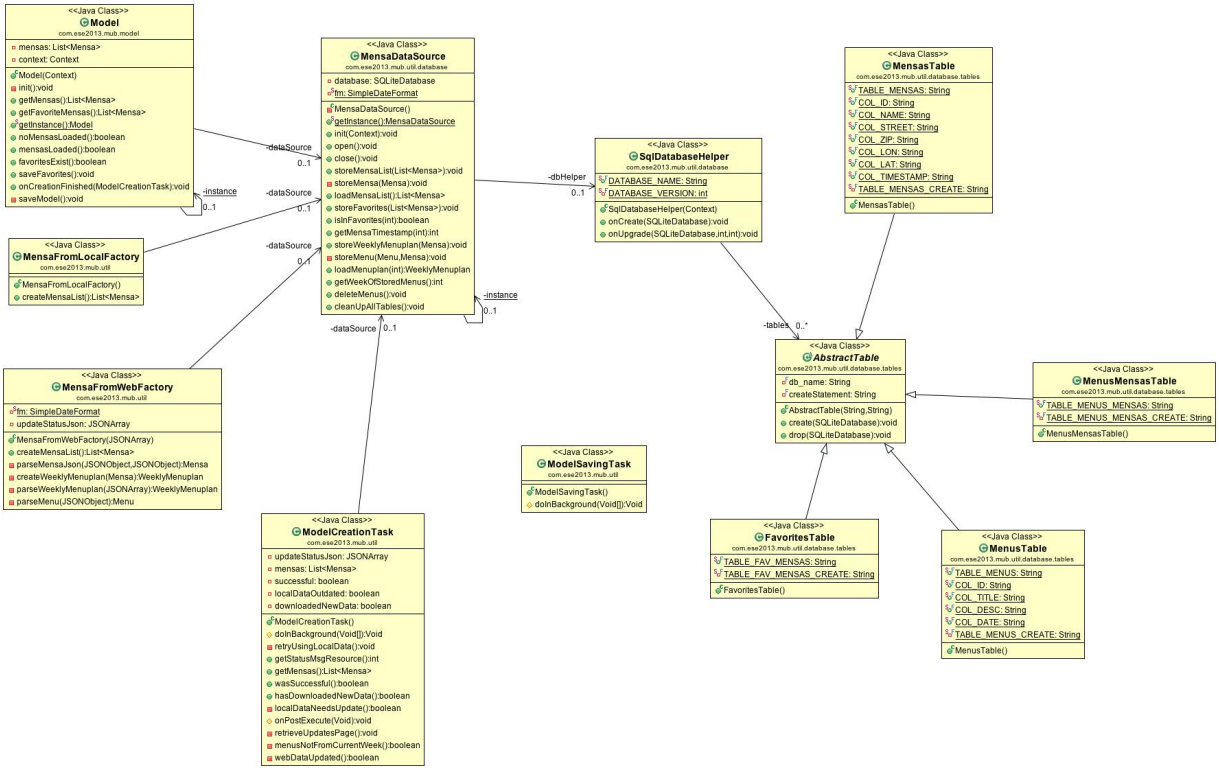
Whenever the user presses the back button on his device, then the application is automatically terminated. This is not a very intuitive behaviour, since the user clearly expects another screen after pressing the back button.
There are several possibilities for improvement:

- Disable the back button
- Display a warning asking if the user really wants to terminate the app.
- If not on the homescreen, jump back to the last screen the user viewed. When on the homescreen, we expect the app to be closed when pressing back.

We recommend the last option since it models the natural behaviour of the back button. For example, when the user is in the mensa list and presses the back button, he expects to land on the homescreen.

2. Strategy to persist Data

This app uses a SQLite Database to persist data. The main access to the databases is given through the SqlDatabaseHelper. It contains the database versions and the behaviour of the database when it is created or updated. Additionally it holds the initialization of the tables. This is the only call for the table objects which have an SqlDatabaseHelper to create the tables. The other class which holds a SqlDatabaseHelper is the MensaDataSource class. This class can open and close the database and give the important methods to handle the data. To make sure that there is only one database helper this class is designed with the singleton pattern. The model class is responsible for storing and loading data of the whole model at runtime. It holds a list of mensas and the method for getting favourite mensas. It can perform the creation and saving task for the model and updates the observer for the gui. The main part for updating the mensas is played by the MensaFromLocalFactory and MensaFromWebFactory. The WebFactory updates the information in the database by using the datasource and creates an actual list of mensas. The LocalFactory loads the data from the database when there is no internet connection so you can see the last updated data. Generally thats a very good way to persist the data. It combines good techniques in object oriented design with distinction of data storing and the model. Maybe there will be a problem with the favorites when the mensa id's on the server will change but this is unlikely.



3. Analyzation of one specific Activity

In this section we will take a look at the main Activity DrawerMenuActivity, which is currently also the only Activity.
Starting at the top of the class, there are a few instance variables where some of them are constants and some of them are UI-elements. It is ok to store the UI-elements here for global access, but you should definately write a short comment for a constant if it is not clear why or how it is used, for example the HOME_INDEX or MAP_INDEX.
Next, lets take a look at the onCreate method. Basically all parts of the UI get created here. It is good that you (partly) bundled the creation of the pieces of UI into a separate build method (like createSpinner). Here and there you could maybe refactor a bit more but in general, we think you did a good job here.
In the createSpinner method: The case structure in the onItemSelected callback looks a bit ugly but we assume this is currently the only way to do this. Maybe you want to refactor this to a private class like you did with DrawerItemClickListener for better readability. Also, make sure you write a short comment why you override this method:

```
@Override
public void onNothingSelected(AdapterView<?> arg0) {
}
```

We assume you did not want the listener to use the super method so you left it empty.

The rest of the methods are relatively small and very understandable. It is good that you did not put any logic into this Activity. Your model stays hidden from this point of view and this is good.