Cross Reviewing

Team 2: Sport@Unibe

Table of Contents

1. Design

1.1 Violation of MVC layers

The application consists the following layers:

- Database (Model layer / Controller layer)
- User Interface (View layer)

All data is persisted on the database only. The UI directly interacts with the database and there's no controlling instance between the View and the Model. It is questionable if it would really be necessary, as the only user input is based on clicking buttons.

The UI always calls the model and the model never calls the UI (i.e. for updating).

1.2 Usage of helper objects between view and model

As the model layer only consists of the database there are a few objects which are used to send information to Activities / Fragments. Those are:

- Sport
    - contains only a name and an ID
- Course
    - contains a Sport and information about time, location, etc.

Those are also needed to tell the database changes of a course or a sport. Each activity gets them from the DBMethodes class. Those helper objects more or less contain what you would expect. Maybe it would be a good Idea to give every sport a List of its courses. Then it is always clear which course belongs to which sport and you don't need to store the sports name twice.

1.3 Rich OO domain model

To draw the domain model you wouldn't need much paper. The main two objects are courses and sports. Courses are part of a sport and you can interact on them by rating, searching the location or setting a timer. The third object is the user himself. He interacts with the courses.

As further objects I could imagine a Timetable of your marked courses, online interaction with friends to invite them to a course…

1.4 Clear responsibilities

The responsibilities of the Java classes are mostly clear after you read the code. They are not documented anywhere (maybe javadoc would be nice). Also the names don't tell you always what the purpose of the class is. You can find more details about the naming in section 2.2

1.5 Sound invariants

No invariants declared in any class.

1.6 Overall code organization & reuse, e.g. views

The Package organisation does not really help understanding the code:

- Why is the ProfileFragment in a separate package?
- What exactly is the difference between the views and the details package?
- Why isn't the Details-Activity in the details package?
- Why is the package with all core elements called 'support'?
- Why are there 2 AlarmReceivers?
- Why isn't the NotificationService in the reminder package?

It is not clear which classes have which responsibilities only by looking at their names. Best examples are JsonSport and JsonCourse who synchronise all the sports and courses.
There is some duplicated code and some Classes could really benefit from the use of inheritance. For further details see 5.4

Just as a remark: Eclipse provides a very nice autoformatting tool for Java code. Some of the classes are just not readable in the way they're formatted now. As an example you can have a look at OptionsActivity.java.
The autoformatting also makes sure, that you don't have to scroll vertically in your code. Just select all (ctrl+a) and then autoformat (ctrl+shift+f).

2. Coding style

2.1 Consistency

Overall the coding style is consistent, but some things are not optimal. For example the Course class has an extremely long constructor and absolutely no setter methods. Maybe it would be better to use setter methods or a factory to define the variables that are less important and not always required, at least this would make the code more readable.

The variable assignment in the constructor is inconsistent in some classes, while most use the this.name = name; way, it is more complicated in other classes, for example in the Date class. Furthermore a lot of the strings used are hardcoded, they should be saved as references in strings.xml instead, this way it's much easier to make changes, reuse strings and to translate the app in other languages.

Why do the Json… classes inherit from AsyncTask? These are always called via execute() and then get(). This makes the async useless, because get() blocks the called thread, so the requests aren't handled asynchronously.

Finally, the code in the Day utility class is very unclear, this class should get rewritten in a way that it is easier to understand for the reader. Also, it uses deprecated methods.

2.2 Intention-revealing names

While most of the class-, method- and variable-names are clear and easy to understand, there are some names that are unclear or strange and in most cases there aren't any javadoc or comments to clarify those.

In the Course class for example, someone who reads the code has no idea what the method "getKew()" does. It returns a string (also called kew), but it is absolutely unclear what it contains and what its purpose is.

Another example of suboptimal naming is the sub(int) helper method in the DBMethodes class. It isn't obvious that sub stands for subscription here, as opposed to the getSub() method in the Course class there isn't any javadoc or comment here.

The names of the AsyncTask classes JsonCoordinate, JsonCourse and JsonSport aren't intention-revealing either. They are used to get data about the coordinates, courses and sports from the internet, so maybe their name should contain that.
Another example for a unclear name is the class DActivity.

2.3 Do not repeat yourself

There are some duplicate classes. Of the ch.unibe.unisportbern.views.details.reminder package, all classes already exist in the ch.unibe.unisportbern.views.details package, the code is exactly the same.

In the class DBMethodes there is also a duplicate method (setRating), the second version has different parameters but is not finished yet, so maybe this could be intentional.

Additional, the sports string array in the CourseFragment class is unnecessary, it would be better to just use an ArrayAdapter<Sport> with the already existing sports ArrayList, therefore the getNames() helper method is not necessary too.

You should also create a super Json class which implements the doInBackground() and convertInputStreamToString() (why is this method static?) methods, because these two methods are the same in every Json… class.

2.4 Exception, testing null values

Generally the exceptions are handled in the code, but it isn't done in an optimal way. There are declarations in the methods that may throw an exception, but they aren't really handled in methods where an exception could occur. All catch-blocks are just empty or only include e.printStackTrace(), and sometimes there isn't even differentiated between the different types of exceptions. It could for example be useful to display a message to the user in some cases or handle them otherwise.

In the executeJson()-method, you throw four exceptions. It would be nicer if you catch these exceptions and then throw a self-made exception. That way, you don't have to catch four different exceptions in the activities.

Null values aren't tested anywhere.

2.5 Encapsulation

For most classes the encapsulation is good. The method visibility is ok, methods that could be used by other classes are set to public, helper methods which are only used within the class are private in most cases.
The encapsulation of the class variables isn't always optimal: in a lot of cases they are private as they should be (or public for final variables), but not always.
For example in the Date, DBHelper or DetailsActivity classes (among some others) the visibility of the class variables isn't even declared and therefore default (package private).

2.6 Assertion, contracts, invariant checks

There aren't any assertions or invariant checks in the code. Contracts with pre- and postconditions only exist in the classes DActivity and SportsAdapter, but even there they aren't checked before and after the method is executed.

2.7 Utility methods

The Network and Day class are the only classes containing an utility method. The first one is used to check if the device has a connection to the internet. It would be better if the method isOnline() is static with the context as parameter, this way it wouldn't be necessary to create an instance of the class before being able to use it.

The Day class is used to calculate the week day from a date. Here too, the method should be static, or maybe even directly included in the Course class, which is the only one that uses it.

3. Documentation

3.1 Understandable

There is not a lot of documentation so we cannot really comment on it. At least the comments that are already in the code are understandable.

For example it could be good to have a short explanation about what the responsibilities of the classes are at the beginning of them (like in the DActivity or the SportsAdapter classes) or some javadoc for methods that are not that self-explanatory.

The UML provided in the SRS is not really helpful. It is not possible to understand the meaning of an object or the relations between them with this diagram.

3.2 Intention-revealing

Sometimes there are explanations where it is not really needed for example in the class Courses there is a little documentation about the getSub() method but we see that it will return the subscriptionRequiered boolean, it would have been good maybe to explain what the String kew stands for.

In the DBMethodes class we have the same comment written twice in the same situation that could be avoided.

3.3 Describe responsibilities

There is not a lot of documentation about the responsibilities except for the classes DActivity, SportsAdapter and ProfileFragmentand and the documentation is good written (you could use more tags if you want to).

3.4 Match a consistent domain vocabulary

Some methods have javadoc but sometimes not that accurate. For example using the tag @param when the parameter referred to is not a parameter of the method (in the DBMethodes class we have @param context for the getAllSport() method).
In the MainActivity it should be @param and not @item:
@item the button that has been clicked
public boolean onOptionsItemSelected(MenuItem item).
There is a good javadoc for the constructor of the TabListener class.

4. Test

4.1 Clear and distinct test cases

The test cases are separated from the main project in a separate project, which is a nice way to program it. The testproject contains three test classes (DBMethodesTest, JsonSportTest and SportTest).
All of those three classes only consist of one method that is tested, which is responsible to test what the whole class stands for. It would be nicer to split the big field of the class-test-responsibility into pieces and write a few methods to test every single feature so you can see which feature throws a error.

4.2 Number/coverage of test cases

At the moment, there are three test classes, which are very skinny. Those three classes are covering furthermore only a smaller part of the main package “ch.unibe.unisportbern.support” and the other 3 packages are not tested (1 package ignored because there was one fragment-class inside). Furthermore is the overall coverage of the package, which is tested, below the number of 70% that is commonly used to show reliability.

4.3 Easy to understand the case that is tested

The test classes are named correctly by what they are testing and the code itself is also easy to understand (see 4.5 Readability). But there are no comments or different methods with specific names (what the method is testing) inside the code, which tells what feature is tested.

4.4 Well crafted set of test data

There are only a few cases tested and so no special cases are considered. Furthermore is the test data often not really reliable tested, it’s often just paid attention to if the data is not null by the method “assertNotNull()”. But of course, there is the possibility that a wrong object or data is created and this would not be detected.

4.5 Readability

Those tests which are written are easy to understand, but that’s more or less the case because they are very skinny. The structure, which is used to write those tests is fine. It includes a setup()- method and an init()-method to before the test starts and a teardown()-method to finish the test. So you can easily see through the code even if the tests will grow during the progress of the development.

5. Persisting Data

5.1 Strategy

To persist the data, a SQLite database is used.The usage follows the Android Data Storage Guide with an implementation of SQLiteOpenHelper called DBHelper and a class DBMethods that provides methods to access the database and for example get all existing sports.
To get all Sports and Courses, a lookup from an URL is neccessary, so an instance of JsonSport or JsonCourse is created. These classes then parse the result from the JSON string and create Sports or Courses.

5.2 Class diagram

The class diagram is pretty clear. There is the main class DBMethods. The activities only need to know this class for persistence. The DBHelper is used for access the database and the Json… classes are used to access the URLs with the JSON information.

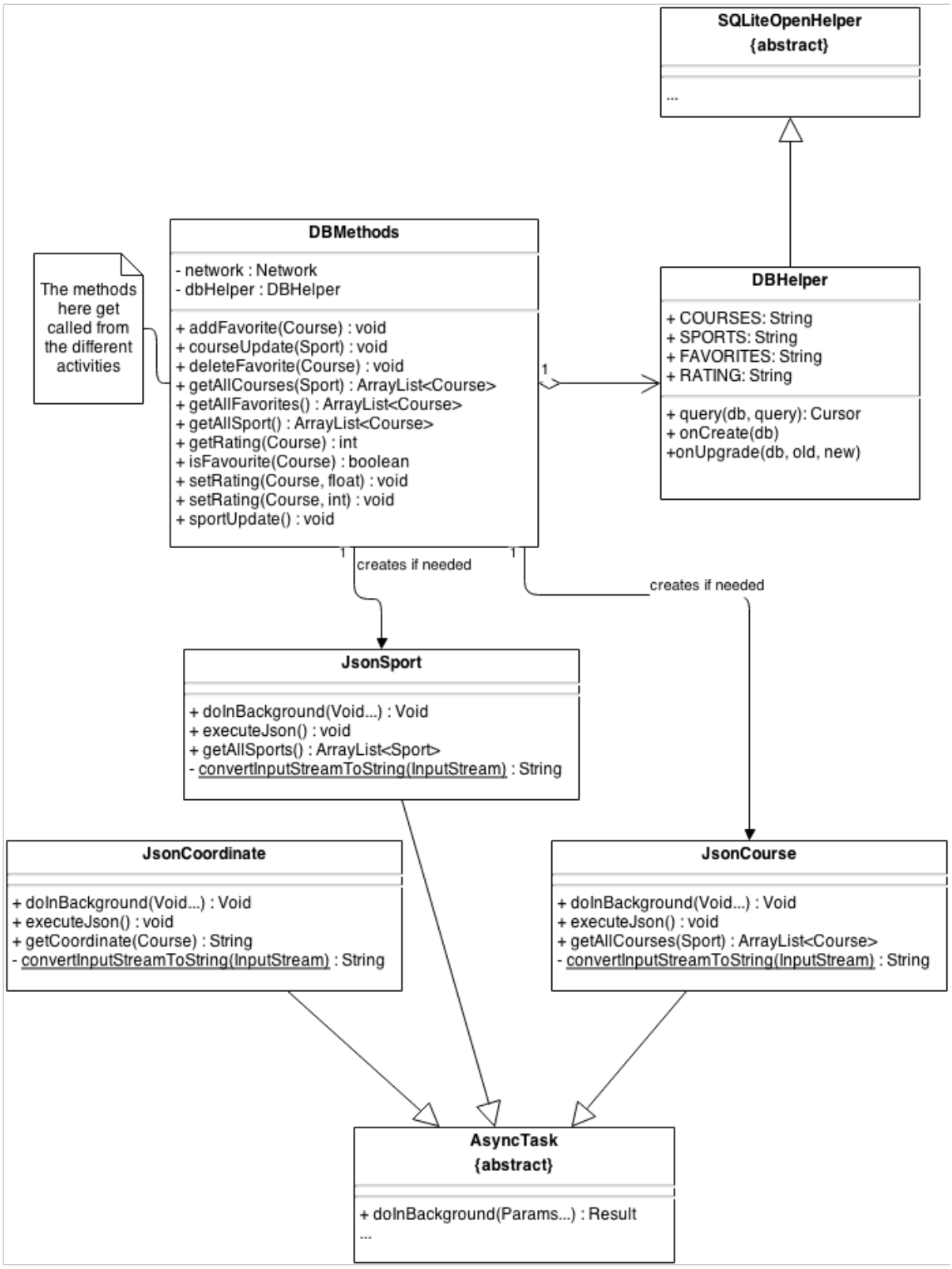
5.3 Sequence diagram

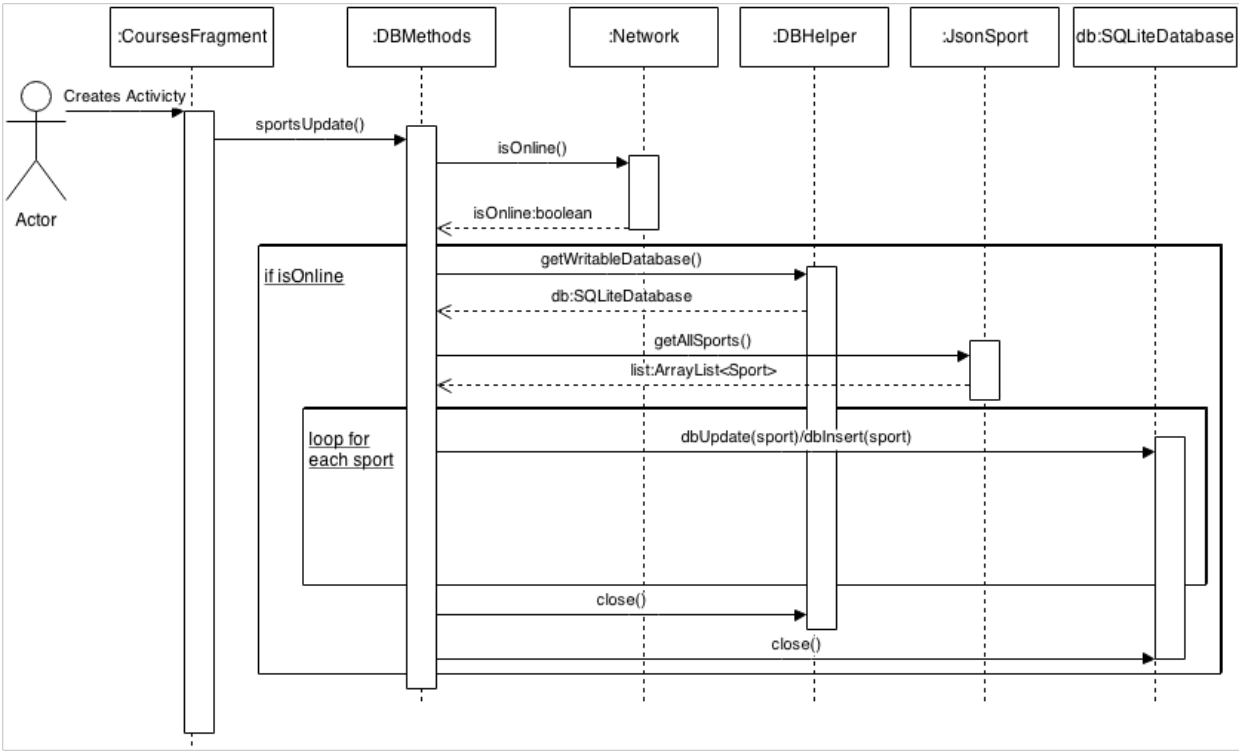
The sequence diagram shows the process of updating the sports.
When the user starts the main activity, it starts the updating of sports. It first invokes the method sportsUpdate() on DBMethods, which is responsible for the whole procedure.

5.4 Conclusion

It's a good idea to have one "main class" for persistence as long as the class doesn't have too many methods. So the different activities have easy access to the data.

To query data from the database, you write your SELECT-queries and put in the WHERE-clause the variables directly ("SELECT * FROM courses WHERE sid="+sid). This is in general a bad idea, because it's open to SQL injections. In this case, it isn't a problem because there is no user input, but you should consider parameterized queries because these are safe for injections (call db.rawQuery("SELECT * FROM courses WHERE sid=?", "" + sid))
It's good that you remember to close the database, but it's save to just call dbHelper.close() because this invokes db.close().


SQLiteOpenHelper
{abstract}
...
DBMethods
- network : Network
- dbHelper : DBHelper
+ addFavorite(Course) : void
+ courseUpdate(Sport) : void
+ deleteFavorite(Course) : void
+ getAllCourses(Sport) : ArrayList<Course>
+ getAllFavorites() : ArrayList<Course>
+ getAllSport() : ArrayList<Course>
+ getRating(Course) : int
+ isFavourite(Course) : boolean
+ setRating(Course, float) : void
+ setRating(Course, int) : void
+ sportUpdate() : void
The methods here get called from the different activities
DBHelper
+ COURSES: String
+ SPORTS: String
+ FAVORITES: String
+ RATING: String
+ query(db, query): Cursor
+ onCreate(db)
+onUpgrade(db, old, new)
1
1
creates if needed
1
creates if needed
JsonSport
+ doInBackground(Void...) : Void
+ executeJson() : void
+ getAllSports() : ArrayList<Sport>
- convertInputStreamToString(InputStream) : String
JsonCoordinate
+ doInBackground(Void...) : Void
+ executeJson() : void
+ getCoordinate(Course) : String
- convertInputStreamToString(InputStream) : String
JsonCourse
+ doInBackground(Void...) : Void
+ executeJson() : void
+ getAllCourses(Sport) : ArrayList<Course>
- convertInputStreamToString(InputStream) : String
AsyncTask
{abstract}
+ doInBackground(Params...) : Result
...


:CoursesFragment
:DBMethods
:Network
:DBHelper
:JsonSport
db:SQLiteDatabase
Creates Activicty
Actor
sportsUpdate()
isOnline()
isOnline:boolean
if isOnline
getWritableDatabase()
db:SQLiteDatabase
getAllSports()
list:ArrayList<Sport>
loop for
each sport
dbUpdate(sport)/dbInsert(sport)
close()
close()

6. Example Activity

MainActivity:

This activity is like the name says the most important activity in the program. It's started when the user opens the program and he almost never leaves it except when he clicks on a course, then he'll start the "DActivity" (tip: name activities after what they are doing). So the activity has a big field of responsibilities, for example it manages the view of the profile and the courses.

But it does that not all by itself, it uses fragment-classes to distribute those responsibilities and so the activity itself does not have too many. That's a nice way to program such an interface and it would be way too overdone if the program would start for each of those views a new activity (it would be slower and wouldn't look so nice).

You can further see it by watching the code of the MainActivity, which is about 70 lines and does not have a lot of code, which is by the way also a good sign that it does not have too many responsibilities. Furthermore it is nice structured and all the public methods do have a javadoc, only the java documentation of the class is missing (and of course the small mistake with the @item in the javadoc).

A thing that we would change is that in the screen of the main activity are some features shown, which don't work. For example the search-function. By pressing on the lens, it happens nothing at all and that's not good for the version 1.0.

One further thing we would change is the usage of the "back button", because when the user switches tabs, and he wants to go back, he'll press instinctively the back button, which on the other hand will close the program. There it would be great to switch back to the previous tab, if they were switched, or else close the program.

But apart from those minor mistakes we can say that the code of the main activity is the result of good work.