1) Review

Design

Violation of MVC layers

The MVC pattern is applied correctly and implemented very clean in this program, as the Model (e.g. the “game” package) is clearly separated from the View (xmls and “gameui” package) and the Content (“board” package).

Usage of helper objects between view and model

Helper object between view and model are used in the gameui-package, for example BoardUI.java, which generates Buttons and keeps these buttons updated as the game continues.

Rich OO domain model

The responsibilities are modelled well; each class has one clear responsibility. Good use of patterns; e.g. observer pattern and template method. (e.g. the getBoard() method in the AbstractBoadGenerator class which is extended by concrete BoardGenerators).

Clear responsibilities

As mentioned before the responsibilites are distributed well among the classes. Each package has one general responsibility, for instance “.game” prepares everything to launch the classes from “.gameui” and “.wordfinder”. The Classes itself are responsible for keeping track of one single part of the app, which is perfect.

Sound invariants

There aren’t many invariants in the code, in fact we only found the one in the “Board.java” classs, where it checks if the size of the board is greater than 0, which makes sense. It would be nice to have other invariants as well.

Overall code organization & reuse, e.g. views

use of deprecated methods ?

Coding style

Consistency
Code is consistent but some times the documentation is not consistent. There are some classes, which have no comments at all and some classes, where every method is commented.

Intention-revealing names
Classes are named very good, only reading their names tells you what they are responsible for. Variables naming makes clear what they are responsible for and what their role is.

Do not repeat yourself
The concrete BoardGenerators are quite repetitive and it is not clear to us, why they are modelled as different objects.It is also not clear why there are so many BoardGenerators when only the PrimitiveDBBoardGenerator is actually used in the code (it is used in the Game class). More documentation here would be useful.

Exception, testing null values
Exception are thrown well and handled well. Null values are not handeled as well as exceptions, but there are fewer places where they are actually handled (for example in Board.java that there actually is a Token being placed and not a nullpointer).

Encapsulation
Mostly instance variables are declared private except where it makes sense to declare them protected. Instance Variables are well encapsulated and only accessible. through set/get methods

Assertion, contracts, invariant checks
There are some asserts (e.g. in Board.java) and some invariant checks (e.g. in Board.java), it is not used very much. But we have to admit that there are not many places in their code where an assert or an invariant check would make sense.

Utility methods
Yes there are utility methods like format() of the Timer class. But there are not too many utility methods.

Documentation

Understandable
If a class is documented then the documentation helps the understanding of this specific class. But there are some classes without documentations like IterativeDBBoardGenerator, SimpleDBBoardGenerator and basically the whole .worldfinder package has very few method comments and none of the class javadoc comments. The comments itself are well written and reveal the function of this method/class in short sentences.

Intention-revealing
The names of the methods are very closely everytime intention revealing, except in the cases when the name is to general, for instance “generate” or “compare”, but We think they’ve done a very good job in method naming.

Describe responsibilities
The documentation describes the responsibility of each class quite well.

Match a consistent domain vocabulary
As far as the code goes there aren’t any inconsistencies in the vocabulary that we are aware of.

Test

Clear and distinct test cases

The Testclasses are named properly, so its clear what they are testing, and the tests are modeled well with clearly structured testmethods.

Number/coverage of test cases

Reasonable number of test cases, most of the important responsibilities are tested. The content of single test cases could be improved as there are functionalities which could get testet in greater detail. For example the test ‘testBoardWithSize1’ case could be extended to an constructor test case, where both constructors of the Board would be tested.

Easy to understand the case that is tested

The methods are also properly named, so it’s clear what exactly is tested. For Example there is a “testSetTokenGetTokenAtCorners” test case, which is a long methodname, but it’s directly visible, what exactly it is testing, and how.

Well crafted set of test data

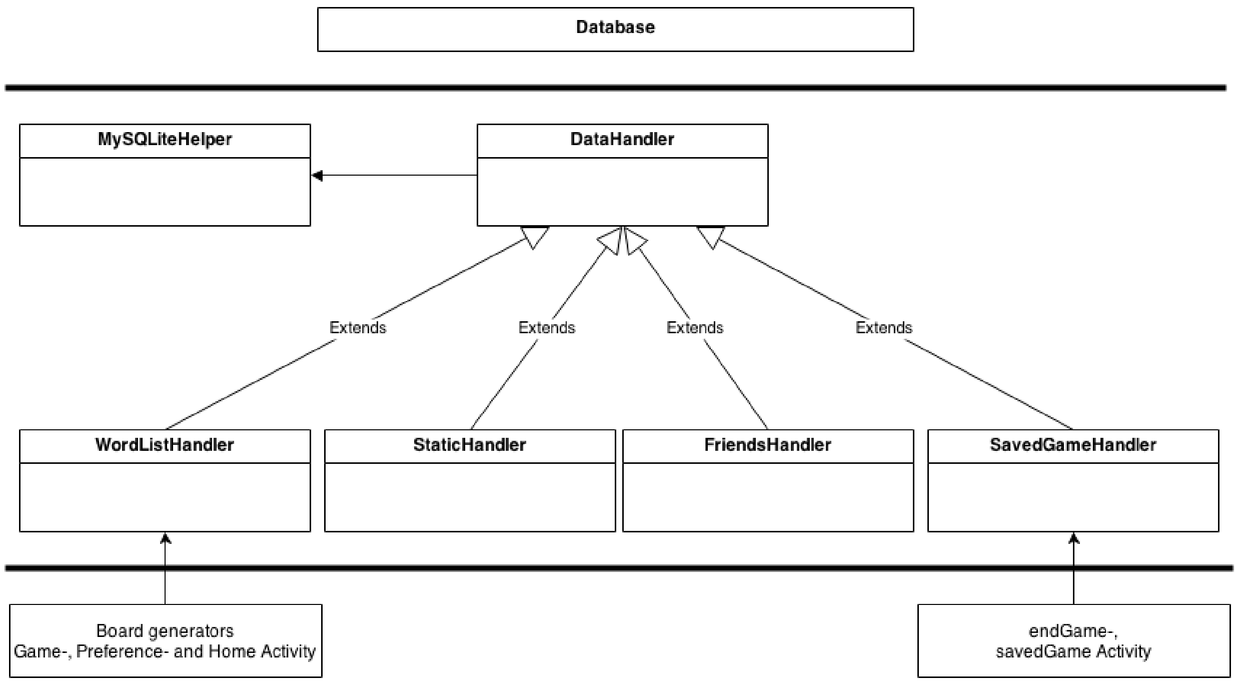
With the extendable Datahandler-Interface test, the connection of the handler tests is promoted. The other tests are structured by their naming, which is appropriate for the size of the test project. The structure could be improved by creating a test suite.

Readability

The readibilty of the test cases is good. The fact that there is an Interface IDataHandlerTest which concrete HandlerTests can extend is well moduled.

2) Data handling

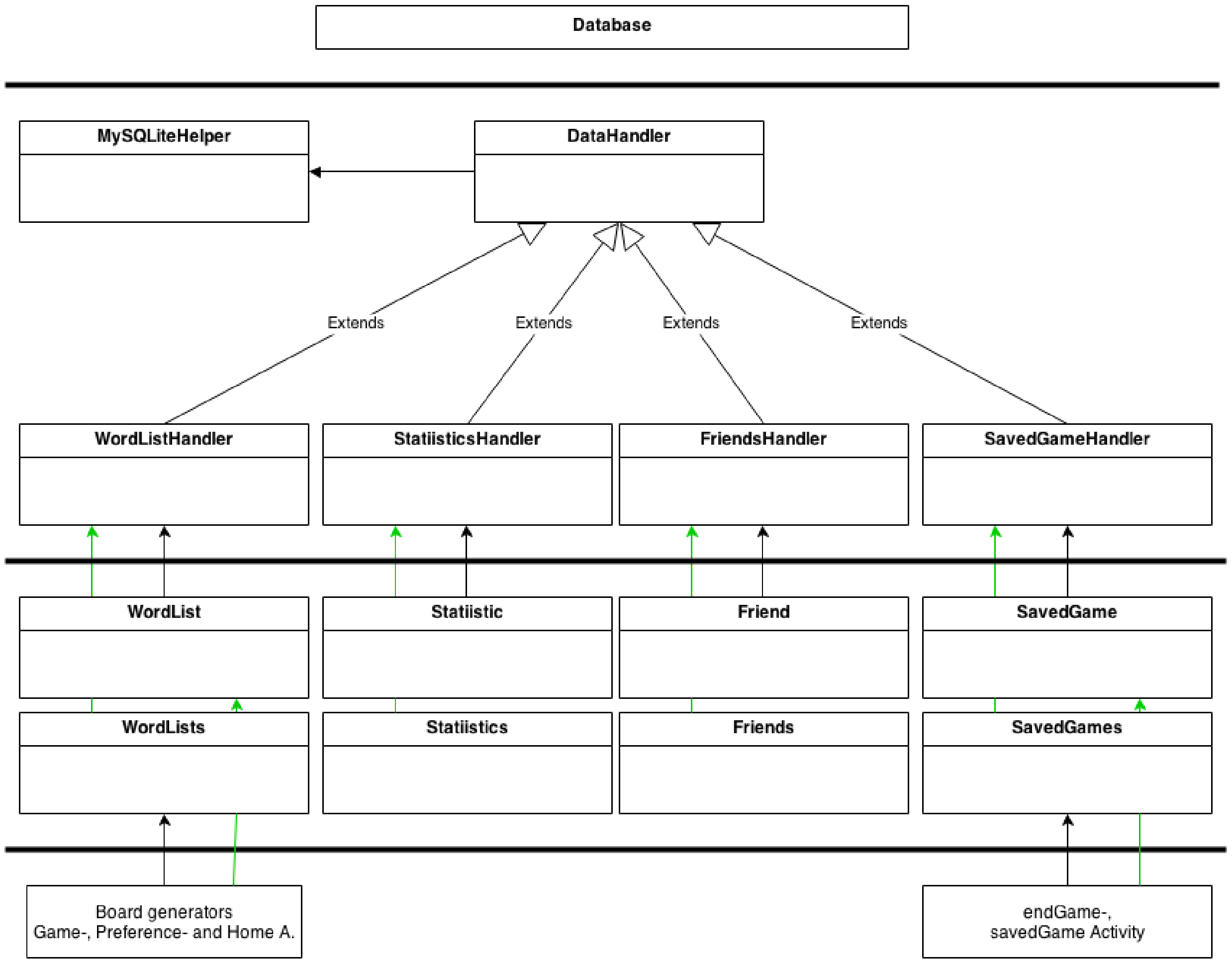
The data processing can be divided into three layers: The database which stores the data, the handlers which provide writing- and reading access to the database and the classes which contain the used data.
This UML provides an overview of the data processing, divided into these three layers:



The nice OO domain modeling of the entire application is also confirmed in the database handling. The reasonable extension of the DataHandler class and the instance of the MySQLiteHelper provide a consistent usage of the database. There is a handler for each table (or table category), which supports the structure in the database itself. Especially cool is the extended Exception.

One major issue that occurs with these three layers is an imprecise distinction between the second and the third layer. Many different Classes (activities, generators, ...) only use some specific methods of the handlers. So most methods of a handler class are designated for a specific class. By adding another layer between the second and the third one, the usage of the handlers could get much more organized.
This layer would contain classes representing the auctual raw data, which was read from or should be written to the database. A get method in the handler of a certain type should return a object of this type (the WordListHandler method ‘getWordlists’ should return a

Wordlist) and an add or update method should take an object of this type as argument (the saveGame method already takes a SavedGame as argument).
Other returntypes such as integers for count-, booleans for exists- or voids for remove methods could be summarized as static methods in a second class. For example the WordListHandler would be accessible by a class WordLists, which contains a static method ‘isWordlistInDatabase(Context c, String wordlistname)’.



Depending on your requirements and further development, this may or may not make sense for you.

One other idea would be to summarize more functionality in the DataHandler class this could help decreasing the complexity of all extending classes.

3) Analyzation of HomeActivity

The Code of this class is short and easy, there are no big code blocks and each Method is linked to by a Button on the Screen.
This Activity does not have too many responsibilities, since all it has to do is redirect to the right place with an Intent, as soon as a Button is clicked.
For example if the user Clicks on “Start Game” it redirects him to the GameActivity and so on.