D7032E Home Exam

Authors
Alex Peschel: alepes-8@student.ltu.se

Understanding software design

October 28, 2021

1 Unit testing

1&2) Bug: The bugs which was encountered for part one and two of the rules and requirements are as follows: the game can have zero players even though the rule is to have at least one; and the board can be set as a 0/0, 0/1, and 1/0 board. All breaking the rules and requirements.

Junit: As for the testing of the bugs it can be seen as something not testable. This is due to the code having no type of response for the settings setting players or board at zero. This is universal through out the code, any place where you would see the amount of players or board size the response is extremely limited if any and nothing testing this specific part. However, there is one way you could see the problem and with an assumption make a junit test. The assumption is that if we insert 0 or any of the invalid values into player and board setting in the class VarietyWordSquares we should get a IlligalArgumentException thrown at us. So if you make code which inserts the values into VarietyWordSquares and expect an IlligalArgumentException you can see of a problem arrives or not. If it doesn't than we know it doesn't work. I did the test with the code in figure 8&9 in the appendix.

5) Bug: There is several ways for a user to stop filling the board such as incorrect input for placement or an empty letter. In these cases the game ends when a certain amount of rounds has gone by

Junit: When reading the code to understand if there is a way to test this i would say no. The rounds is already decided on in before hand and not depending on the empty spaces which are left. This can be seen in the figure 10 in the appendix. So in turn the progress doesn't care about the game and what we insert. Testing this would not return a value, this includes a board or player which we could check to see how the board looks at the end of the game.

8d) Bug: When inserting the same word twice the word is counted both times.

Junit: The unit testing for this can be done with the help of calculateScore within the class. First of all you need to create an instance of the class VarietyWordSquares and stop the loop. Which I do with a function called provideInput. Than there will be a creation of arrays so that one can store the desired word and then send that into calculate test. The code within figure 11 will show this. And so that provideInput works the code in figure 7 in the Appendix.

8e) Bug: It shouldn't be wrong however this is something which does go wrong due to rule and requirement 8d.

Junit: The bug follows somewhat the same problem as to 8d, where the same word counts twice instead of once. Then on top of that, the normal settings for scoring does not provide words with more score however you make the word. So every word gives a already determined value, due to it's length. As a result this can not be tested due to all the already aforementioned parts.

10) Bug: When getting equal scoring with someone there will still be a winner.

Junit: This can not be tested. The value is only printed out for the user to see and not returned do to the class being a void class. It would also be unreasonable to suspect an error. The parts related to printing out the winner can be seen in figure 12.

2 Quality attributes, requirements and testability

What requirements do you have to value when a software is easy to modify and extend, it is all relative to the implementation and user. On top of that, as a second sentence is spelled out it sounds as if the most important part is to implement the features in 13-17 while perhaps not focusing on modifiability and extensibility. Then when we talk about part 19, the text implying different phases of the gameplay can be expressed as testability. However, testability focuses more on finding out which software component are of poor quality and what's wrong with the code on a deeper level. The phase and gameplay is more centered around software testing.

On part 19, saying the Testability quality attributes when implying testing different phases of the game-play can be expressed as somewhat incorrectly. Testability focus more on finding out which software components are poor in quality and while software testing checks the game-play so that things works as it should. Testability finds problems which can hide from software testing.

3 Software Architecture and code review

3.1 Extensibility quality attribute standpoint

Extensibility is the system design principle where the implementation takes into consideration the future growth of the program. So for high extensibility the program should be considerate of future changes to the program. That is something the program at the moment isn't.

So for the extensibility of the class we can take into consideration that the file has classes with low primitiveness and low cohesion in it. This makes it somewhat harder for future changes to be made. When checking one class one can see that it has a large amount of functions and functionalities to it instead of just one. We could take player as one of the best examples. The player class places letter, read messages, picks letters and send messages but also store information in regards to color of the board. As one can see the player class has a bunch of different functions in it which supports the low primitiveness. Then for the cohesion, players could be a good example to that as well. As I said before the players stores information about the board and the player. Which lowers the cohesion and in turn does not support future extension of the code. For better design which supports extrensibility the code should improve on the primitiveness and cohesion. Primitiveness so that one class can be easily used by new and other functions and cohesion so that the files doesn't work with more than one type of information such as the board.

3.2 Modifiability quality attribute standpoint

Modifiability is the easiness or hardness to change and modify a system. Compared to extensibility which is the coding so that it will be to extend and modify the code modify is how easy it is to actually change the code and add new features.

As a good start to see if the code is easy or hard to modify is to compare with the feature requests given to us. So if we took and looked at request 13 for a possible addition the code would at the current moment make it hard for the coder to implement the desired request. Especially the part regarding a shared board. For the implementation to be added the coder would need to both edit existing code and add already existing code with different functionalities set for the shared board. For example we would need to add a new game class and player class so that they don't create their own boards. The overall underlining information is that the code has a high coupling and low cohesion to it. For more example to why the code has low cohesion go to part 3.1. However, for the coupling part we can see that for example the player class has high coupling to other classes such as the server and game. Which makes it harder to change because this would effect other classes other than the intended one. For some reason the player creates and uses connection commands even though there is a class called server with just that purpose.

3.3 Testability quality attribute standpoint

Without suitable code the process of testing the system becomes harder and many functions can only be tested through calling them from somewhere else to see the results.

The application as it is does not support the process of testing in a friendly way without a large amount of work or changes. One underlining reason for this is that a large amount of classes does not return a value or have any indication that something has gone wrong, such as IllegalArgumentException. On top of that we can relate to the fact that classes has many functionalities to is making it harder to test a specific part of the code.

4 Software Architecture design and refactorin

The flow of the game goes as something simular to the flow chart below in figure 1. The color represent which package this is mainly done in. Options is written out by BoardUI, but it is called upon by GameHandler, so i catagorize it as it happens in GameHandler package. Colors seen in figure 2

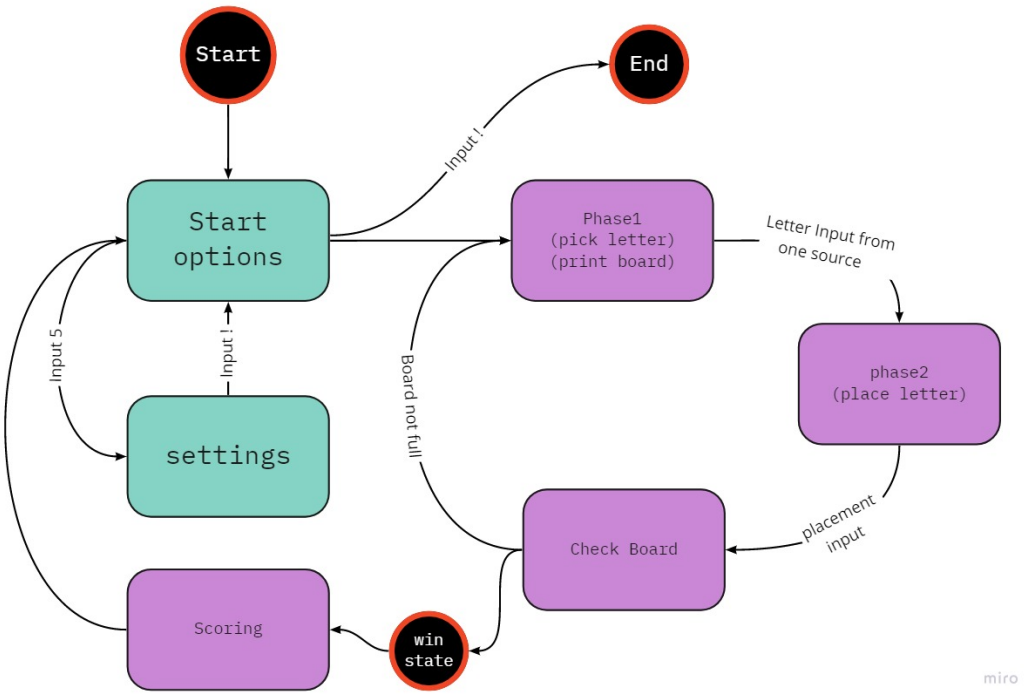

Figure 1: Flowchart

4.1 Modifiability

The code which was given to us had several parts which could be improved with modifiability in mind. One part is to split up the code into separate classes and into different packages. This way the dependency's on other function and classes has changed and as a result circular dependencies has been taken out of the equation. Instead, as one can see in the figure 2 below, each package has either one or more dependency's while none goes back to the one calling it. This has been represented by levels and packages only call on lower levels.

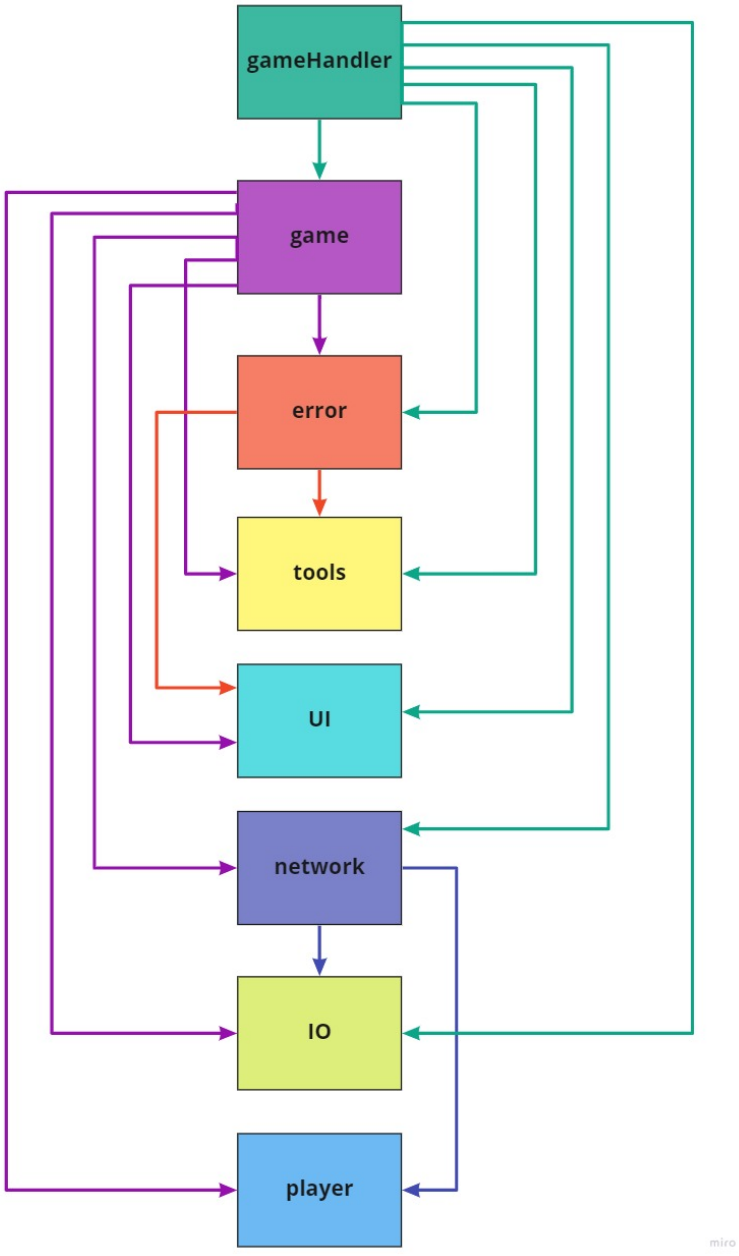

Figure 2: Package Diagram

The result of the new structure is that the coupling within the system has been lowered. On top of that, cohesion has been raised which is a good for modifiability

The cohesion has been a large focus for many additions to the code. With cohesion in mind several functionality has been split up into the smaller parts which only focus on one area. For example of the opposite would having been if a function or method within a class handles both logic and UI. One example of this within the given code would have been within the function VarietyWordSquares. It had an assignment to handle the menu and depending on the input call on different game modes. However it also create boards, initiated games, UI and scanner all within one function. To improve the cohesion each part is split into independent separate classes handling one area each. Then letting the GameHandler handle the communication between the classes. This can be seen in the figure 3

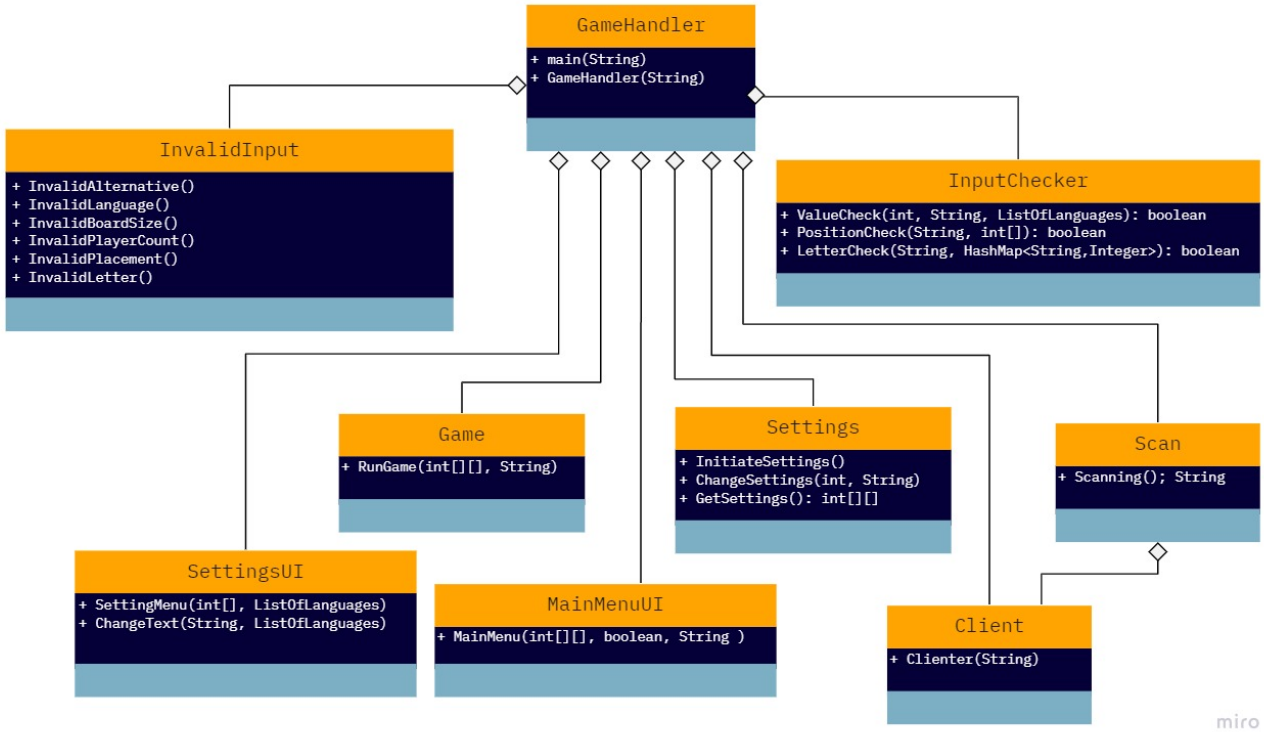

Figure 3: GameHandler

As a result of the focus on cohesion the primitiveness and completeness has also been improved as well as the possibility to reuse parts of the code such as SettingsUI.

If one would look at the code within the class VarietyWordSquares one would see that within one class there is several different functions as well as different functionalities within those functions. The completeness is improved with the help of localising specific types of data into the relevant classes. If one would look at the player function with the VarietyWordSquares class one could see that player is storing colors for tiles, which has nothing to do with the player. So to improve the completeness this has been changed so that for example the color is separated into the BoardUI, which is the only one who needs that information. And then the player only stores information relevant to the player such as ID and their own score. The new system regarding players can be seen in figure 5 and the BoardUI can be seen in figure 4.

4.2 Extensibility

To improve the extensibility of the given system there is several parts of the code which needed to be changes.

One of the first things which has been changed and created was an abstract game mode, which I named AbstractMode. AbstractMode has two functionalities to it. Firstly it has the GameProcess and the other one is CalcualateScore. The GameProcess is a function which calls upon the sub classes PlacePhase and PickPhase. These being divided up separately from the AbstractMode class for the ease of adding new Phases in the future. Then the CalculateScore within AbstractMode is a function which takes out the score as well as checks if there's a winner or not for the game. This has been illustrated in figure 4

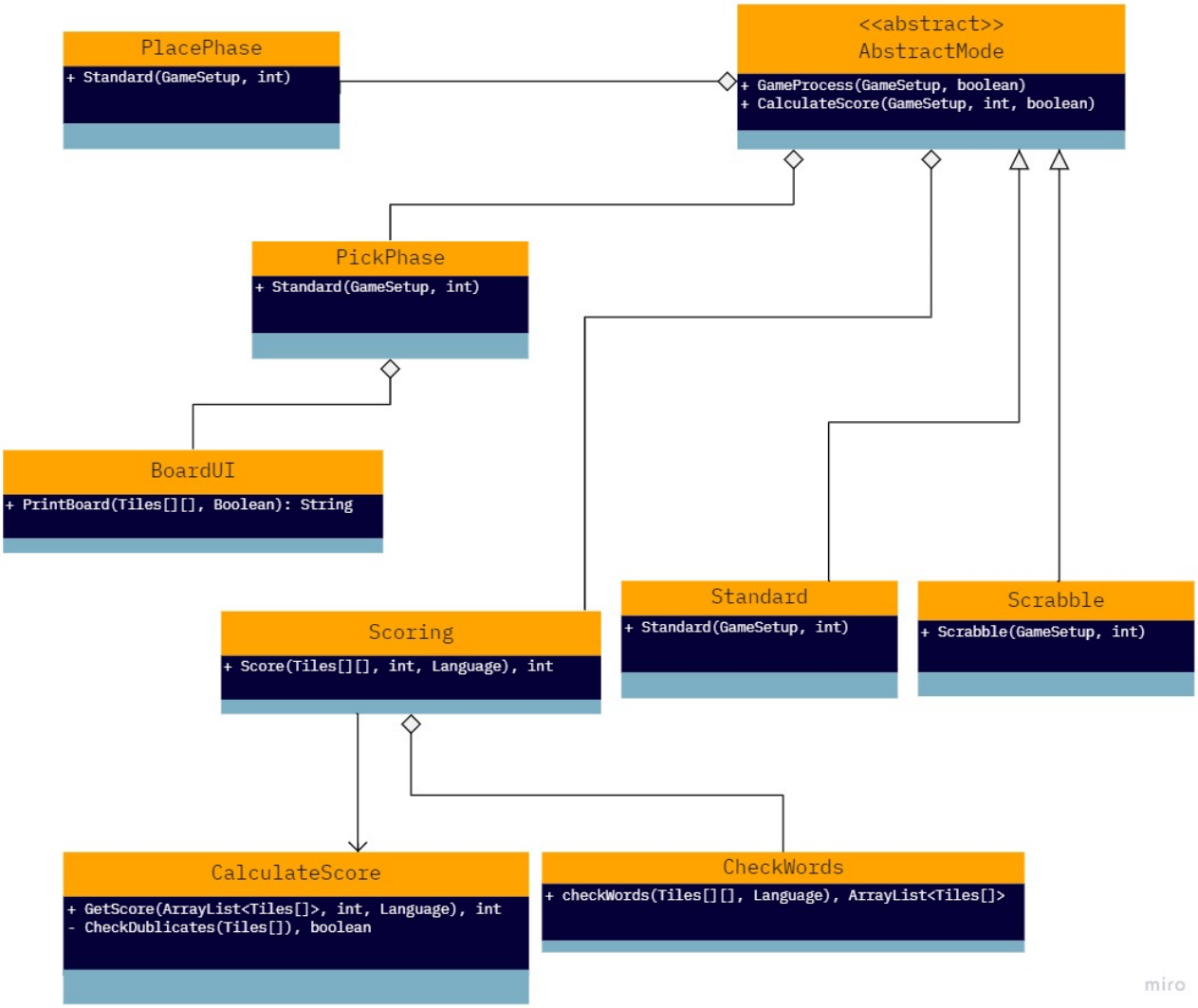

Figure 4: Abstract GameMode

In this part, as shown before, there is few dependencies between classes. The AbstractMode is inherited by two sub classes, as one can see in the figure above. This has been done so that code isn't rewritten unnecessarily for every mode, but rather reused. Then if one mode needs to change something they can override the inheritance with their own GameProcess or CalculateScore. The goal with this was improve the possibility to add new game modes with ease.

Then as a second part the GameSetup package which has a class called GameSetup. This class manages the setup of different parts of the system such as creating the dictionary with every word in it for scoring or the player instance for every contender. However, every part separate from GameSetup in the figure can stand independently and in turn be reused for other systems.

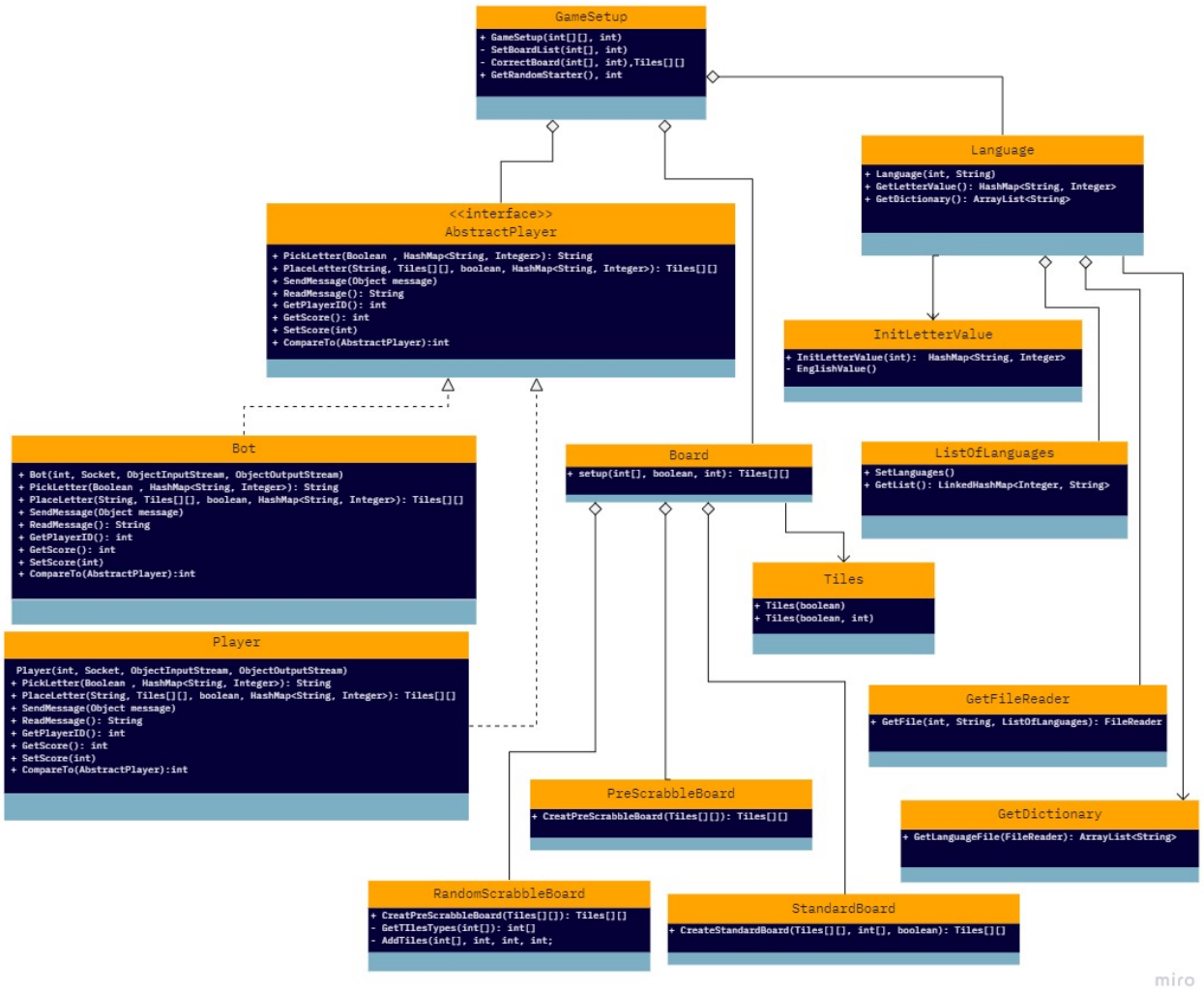

Figure 5: GameSetup

Having the extendability in mind I created the Board class which has several different options for creating a board in separate classes. This is so that Boards only assignment is to creator the initial board and then call on the needed classes depending on the game mode. This is done instead of having every board creation within Board class. In that case, the class would become unmanageable when adding a couple more options. However, the AbstractPlayer class were in focus of the extendability and re-usability. I created an abstract player class as an interface. This way I have defined functionalities for classes implementing it. With the interface class the goal is to ease the implementing of new player types, while not needing to read though to much code. Especially if every player mode were included into one class, this would result in many if statements. Resulting in unreadable code. The only part not in the interface is the constructor, however the only part it's there for is to insert the players initial values such as id or online status. As can be seen in figure 5 there is a separation for language as well. This is to improve the possibility to include more language into the game. I create a language class calling onto different part with all different functionalities within the same area. I decided to divide it into smaller parts called ListOfLanguages, InitLetterValues, GetFileReader and GetDictionary so that if one wishes to add a new language with or without new letter values it should ease that addition. With the coding one only needs to work with InitLetterValues and ListOfLanguages in a minimal ways.

4.3 Testability

As the code is divided into smaller sub-classes, such as phases of the game process, it becomes easier for the user to do unit testing. This is because you can test more specific part of the system instead of needing to go though unnecessary code. This can be seen with the PlacePhase for exemple. During the test we don't need to do anything with the AbstractMode, but rather only the PlacePhase class it self.

5 Re-engineering the code

5.1 Testing

The testing is located in package called test within src, and has several class for different functions to ease the possibility to test and customize the code.

Figure 6: TestPackage

5.1.1 Prerequisites

To be able to start the test you will need to change the "String path" value within the "TestWordSquare" file. The new value should be your own direct path to the src package.

As for now, when testing, the code thinks the absolute path to the src package is completely different than the actual direct path. So a solution to this is to pick out the absolute path by your self. The path should look something simular to **"E:/skola/code/WordSquare"**, where WordSquare is the folder holding the entire game. This can be seen in figure 6. Make sure not to end it with a slash.

6 references

- https://www.generacodice.com/en/articolo/133943/JUnit:+How+to+simulate+System.in+testing%3F
- https://stackoverflow.com/questions/23653875/how-to-simulate-multiple-user-input-for-junit
- https://stackoverflow.com/questions/10575624/java-string-see-if-a-string-contains-only-numbers-and-not-letters (20/10-21)

A Unit Testing Figures

```
private final InputStream systemIn = System.in;
private final PrintStream systemOut = System.out;

private ByteArrayInputStream testIn;
private ByteArrayOutputStream testOut;

@Before
public void setUpOutput() {
    testOut = new ByteArrayOutputStream();
    System.setOut(new PrintStream(testOut));

}

private void provideInput(String data) {
    testIn = new ByteArrayInputStream(data.getBytes());
    System.setIn(testIn);
}

@After
public void restoreSystemInputOutput() {
    System.setIn(systemIn);
    System.setOut(systemOut);
}
```

Figure 7: provideInput setup

```
@Test (expected = IllegalArgumentException.class)
public void enoughPlayers()  {

    String testIn = "5" + System.getProperty("line.separator") + "3" + System.getProperty("line.separator") + "0";
    System.setIn(new ByteArrayInputStream(testIn.getBytes()));

    VarietyWordSquares testUnit = new VarietyWordSquares(new String[0]);

}
```

Figure 8: enoughPlayers

```
@Test (expected = IllegalArgumentException.class)
public void boardSize() { /* The game-board is a grid with at least one row and at least one column*/

    String testIn1 = "5" + System.getProperty("line.separator") + "1" + System.getProperty("line.separator") + "0/0" ;
    System.setIn(new ByteArrayInputStream(testIn1.getBytes()));
    VarietyWordSquares testUnit = new VarietyWordSquares(new String[0]);

    String testIn2 = "5" + System.getProperty("line.separator") + "1" + System.getProperty("line.separator") + "0/1" ;
    System.setIn(new ByteArrayInputStream(testIn2.getBytes()));
    testUnit = new VarietyWordSquares(new String[0]);

    String testIn3 = "5" + System.getProperty("line.separator") + "1" + System.getProperty("line.separator") + "1/0" ;
    System.setIn(new ByteArrayInputStream(testIn3.getBytes()));
    testUnit = new VarietyWordSquares(new String[0]);
}
```

Figure 9: boardSize

```
public void game(int startPlayer) throws Exception {
    Player currentPicker = players.get(startPlayer);
    int rounds = currentPicker.board.length * currentPicker.board[0].length;
    for(int i=0; i<rounds; i++) {
```

Figure 10: rounds

```
@Test
public void standardScoring(){    java.lang.AssertionError: expected:
    provideInput("!");
    VarietyWordSquares testUnit = new VarietyWordSquares(new String[0]);

    words = new ArrayList<VarietyWordSquares.Square[]>();
    ArrayList<VarietyWordSquares.Square> words1 = new ArrayList();

    VarietyWordSquares.Square square = testUnit.new Square(false);
    square.put("T");
    words1.add(square);
    square.put("E");
    words1.add(square);
    square.put("S");
    words1.add(square);
    square.put("T");
    words1.add(square);

    words.add(words1.toArray(new VarietyWordSquares.Square[words1.size()]));
    words.add(words1.toArray(new VarietyWordSquares.Square[words1.size()]));

    assertEquals(2, testUnit.calculateScore(words));    Expected [2] but was [4]

}
```

Figure 11: scoring

```
Collections.sort(players);
String winnerMsg = "Winner: PlayerID "+ players.get(0).playerID+ ", Scores:\n";
for(Player player : players) {winnerMsg += "PlayerID " + player.playerID + " Score " + player.score + "\n";}
for(Player player : players) {player.sendMessage(winnerMsg);}
System.exit(0); //quit game
```

Figure 12: Winner scoring