D7032E Home Exam:
Variety Boggle

Axel Alvarsson
axealv-6@student.ltu.se
Department of Computer Science, Electrical and Space Engineering

October 25, 2020

Contents

1 Short Introduction

The content of this course has been very useful and I would dare to say that I learned more than what could be shown in this project. Java has never been a fun language in my opinion and this is the first time writing an actual application with it, but i took this home exam as a chance to learn more as I developed. Given the opportunity of a ”do-over", I would use another language such as Go-lang. And if there was time for iterating more over the current code, some parts could achieve a higher degree of modifiability.

2 Unit Testing

- Requiremnt 1: Not currently being fulfilled as one can set players to lower than 2. As there is no check what so ever for the amount of players, unit testing will not be possible.
- Requirement 2: Randomizing the board is currently being fulfilled. One can test this by testing the randomizeBoard method with the standard *boggle16* variable and ”assertNotSame" against the output of the method.
- Requirement 3: This is currently being fulfilled as one can not enter a word with dices not adjacent. A test can be written for this, although you need to spoof a randomized board and an input. Then with an ”assertTrue" the word input adjacency can be verified.
- Requirement 4: The winner is currently not being announced at the end. This writing a test for this would not be possible.
- Requirement 5: This is being fulfilled as the length of the input equals the score of the word. Writing a test for this would be possible, add words to player word list and assert the expected result against what ”calculateScore" outputs.
- Requirement 6: This is being fulfilled by the current code. Using the same method of testing as with requirement 5 one could test this.
- Requirement 7: The current code provides support for this. Without using a larger part of the system, one can not unit-test this requirement. A test would require player creation, which is bound to the startup of the game with a lot more dependencies.
- Requirement 8: As only one dictionary is currently implemented, and no function for switching dictionary is implemented, this would not be unit-testable.
- Requirement 9: No logic exists for printing all possible words for the previous game. Thus unit-testing would not be possible.
- Requirement 10: This is possible in the current code. Since the choice of board size is bound to starting the game and waiting for user input, ”unit"-testing is not possible.
- Requirement 11: As no other languages is supported and thus no dice configuration exists for them, unit-testing would not be possible.
- Requirement 12: Currently fulfilled in the project. Just as requirement 3, spoofing a board and an input with the correct flag into ”checkWord", one could test once in adjoining chain with an assertTrue.
- Requirement 13: This is currently being fulfilled as geneours is handled in the searching methods. Same as the previous requirement this is possible to test. Although it goes without saying that one would need to be confident in that the coupled ”checkWord" and ”search" methods are correct.
- Requirement 14: This is currently being fulfilled. Assuming a new mode could be implemented and it, structure wise, searches inputs in the same way. Then it could be tested by an approach similar to requirement 13.
- Requirement 15: The code provides support for this. Without instantiating players, which would require running the game, this is not testable. Certainly not unit-testable since this required lots of dependency methods to run.
- Requirement 16: Currently, this is supported by the code. Without running the game one could not test this. Same as previous requirement this cant be unit-tested because of method coupling.
- Requirement 17: This is currently not fullfilled by the current code. No logic is in place to check for valid math symbols, thus not testable.
- Requirement 18: Without re-implementing the code, one could obviously not test this requirement.

3 Quality attributes, requirements and testability

Why the code given by *Boog*, concerning modifiability and extensibility, is poorly written, trickle down to the lack of though when designing the overall structure of the project. Since we look for a design that provides the opportunity of future growth and modifications, hard coding values and function procedures, as this code does, fully removes the possibility of adding new functionality without re-implementation or re-coding large portions of the project. As testability goes for *Boogs* code, it is a mess. Testability depends heavily on modularity and good structure design since they prove more suitable for systematic step wise testing, rather than this severely monolithic and unstructured program. Testing individual parts can only be done in a few places in this heavily coupled code.

4 Software architecture and code review

A good starting point for general overview of the design is *sufficiency* and *completeness*. Regarding sufficiency, the classes captures way more than is sufficient for that class. The best example of this is the *VarietyBoggle* class which contains a large amount of functions that seem to not belong there.

As for completeness, the behavior that is modeled is fundamentally hard coded which leaves no room for re-usability, if future implementations are made. The thing that one might argue goes in the favor of this code, in the scope of completeness, is that this code takes no time to implement. Yet this quick implementation diminished in the light of everything else wrong with the design.

4.1 Extensibility

From an extensibility standpoint, the code is poorly written mainly because of the lack of though throughout the design and overall structure of the project. Since we look for a design that provides the opportunity of future growth and modifications, *Boogs* approach of hard coded function procedures fully removes the possibility of new functionality through small modifications of existing functions. For example, the current code has interdependence throughout most of the system which makes for the opposite of a high priority goal of extensibility, low coupling.

4.2 Modifiability

As modifiability goes for the current design we are hoping for readable code with clear dependencies. From the start we notice a fallacy in the current code which goes against the modifiability design principle, the size of a module should be kept concise. Aside from the obvious, the system does not provide any documentation for extending the program and even if it wanted to, a full refactoring would be necessary to provide for low coupling and high cohesion.

The degree of coupling in the current design is difficult to asses to anything other than *full coupling*. Module wise there is two, where one contains the entirety of the game and the other the client side connection. As coupling generally goes hand in hand with cohesion an assumption that the cohesion should be low would be correct. The *VarietyBoggle.java* file contains everything from messages to thread creation. This makes comprehending what goes were difficult to say the least, and in the scope of modifiability it's next to impossible without a major rehaul of the project.

4.3 Testability

With the state of the current code being primitive to no extent, as well as the modularity problems mentioned in Modifiability, the degree of testability is alarmingly low. A well designed, structured and modular program are often suited for systematic step wise testing. Since the current code is unstructured and rather monolithic, the possibility of achieving high percentage testability is obviously low.

For example, if we wanted to test specific parts of this program, say the creation of threads and their execution, we would be forced to go through the entire game play and somehow extract the thread tasks for analysis. This leaves for a lot of untestable areas in the code which, if were testable, could have displayed possible critical errors.

5 Software Architecture design and refactoring

When reviewing the requirements and designing the refactored problem the major choice of design, for me, came down to the game modes and future implementations. Some presumptions had to be made as to which extent the system should be extendable and modifiable concerning game modes. I opted for extendability withing the general range of what the original variety boggle game mode is. In other words, some setting options and game procedures are coded so that future game modes should comply to them. This was done by using a *.json* format to store the game mode settings and by doing so we can minimalise the effort of implementing new modes. Opposed to keeping every mode in its own class, this design choice does not require the need for writing new rule-checkers or basically copying code for every new game mode.

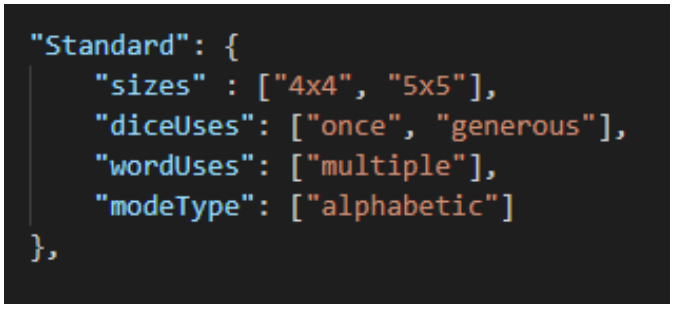

Figure 1: Example of game mode json design.

Although, this format is not perfect. Unlike the new-class based design we notice that the extendability reach is not as great. Another reason for choosing *.json* data structure format, was to simplify mode addition or modification to the degree that a developer might not be needed. That is, a user with a little cunning could easily implement a new mode as long as it is reasonably different.

Aside from the restructuring regarding quality attributes and design patterns, another problem was the performance of the project. Use of memory and memory access could be handled by good refactoring and java's built in garbage collector. But the in game search process for user-input was in dire need for re-implementation. Instead I opted to rely on a *Trie Node*-binary search tree algorithm with which I could find all possible, letter wise adjacent, words on a given board and store it in stead of the dictionary file. Then instead on linearly comparing the user-input against the large dictionary, I could simply compare the input against the relatively tiny list of possible words without the need to check for adjacency.

5.1 Quality attributes

Most of the project should adhere to the primitive quality attribute as much as possible. Any operation a project component can perform, another class should be able to implement painlessly. By this design the classes will be kept small and easy to understand.

A good example that provides for low coupling, high cohesion, and real primitiveness is the abstract player class. The player object specific methods, and the class itself, adhere by default to the quality attributes by following the standard *setters and getters* design in OOP. The methods in the abstract class and all other classes should by the design stand by no interdependence, thus having low to no coupling, and belong together which provides high cohesion.

The methods for the start up sequence of the project will be burdensome to keep with low coupling. But even though coupling and cohesion generally goes hand in hand, by the design choice of keeping the connected start up methods in one class, whilst only accessing static methods in another class, the degree of cohesion between elements in the module will still be high.

As shown, the directory structure in figure 2, the refactored design holds game essential components separated but the cohesion between them high. Every module should provide sufficiency or their less valuable parts should be moved into another suitable class or discarded.

The testing suite built for requirements $1 - 18$ does not interfere with the code itself, but for the test to reach the methods, some visibility had to be set as public even though it should be class or package private otherwise.

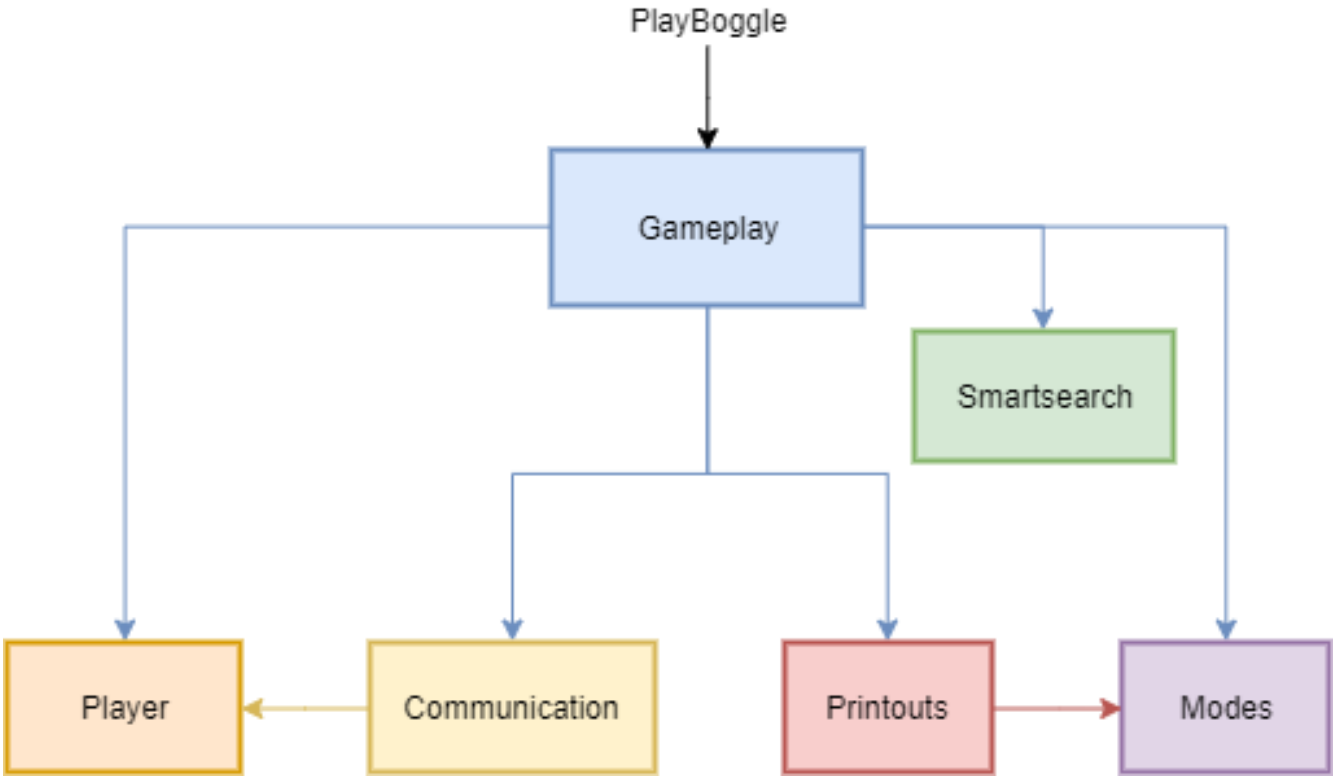

Figure 2: Directory diagram

5.2 Design patterns

The refactored design mainly aims for the *Object Pool Pattern* which is heavily featured in the *GameController* class. The **Object Pool Pattern** which is performance related. Since the creation of objects is costly, reusing the object and keeping it in the object pool, or in this case the *game controller*, for when it is needed reduces performance cost. As shown in figure 2 and figure 3, the game controller acts as this object pool as it contains the active objects when the game is running. What makes this approach useful is the performance boost when object creation might be high. Using this design pattern is most effective knowing that we might have a large amount of clients asking for the same resource at different times in one game.

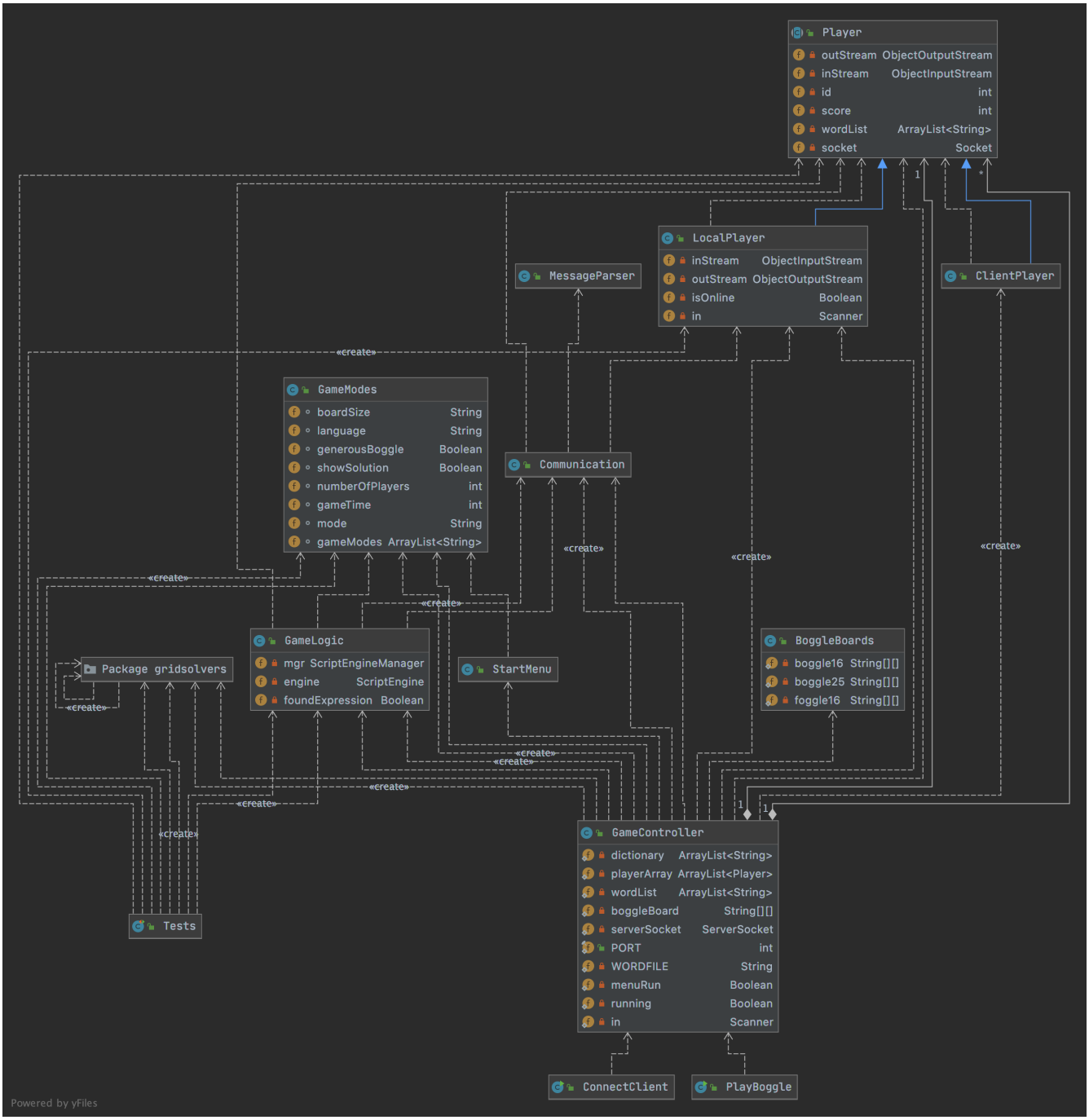

Figure 3: Uml diagram

6 Quality Attributes, Design Patterns, Documentation, Re-engineering, Testing

6.1 Prerequisites

- Using IntelliJ is strongly recommended to run the test and the code with ease.
- A Java JDK 14 is recommended but JDK 11 should suffice.

6.2 Build and run

This project is build in IntelliJ and can be compiled using Maven.

6.3 Testing

Testing suit is easiest to run via IntelliJ. This provides a sequence from which passed or not passed test can be viewed.

References

[1] GFG: Trie, Insert and Search,
`https://www.geeksforgeeks.org/trie-insert-and-search/`

[2] FreeFormat: Regular expression tester,
`https://www.freeformatter.com/java-regex-tester.htmlad-output`

[3] Niklas Johansson Anton Löfgren: Designing for Extensibility
`https://rb.gy/y95wbx`

[4] UML Class Diagrams Reference
`https://www.uml-diagrams.org/class-reference.html`