Home Exam – Software Engineering – D7032E – 2020

Teacher: Josef Hallberg, josef.hallberg@ltu.se, A3305

Instructions

- The home exam is an individual examination.
- The home exam is to be handed in by **Friday October 26, at 10.00**, please upload your answers in Canvas (in the Assignment 6 hand-in area) as a compressed file (preferably one of following: rar, tar, zip), containing a pdf file with answers to the written questions and any diagrams/pictures you may wish to include, and your code. Place all files in a folder named as your username before compressing it (it's easier for me to keep the hand-ins apart when I decompress them). If for some reason you can not use Canvas (should only be one or two people) you can email the Home Exam to me. Note that you can NOT email a zip file since the mail filters remove these, so use another compression format. Use subject "**D7032E: Home Exam**" so your hand-in won't get lost (I will send a reply by email within a few days if I've received it).
- Every page should have your full **name** (such that a singular page can be matched to you) and each page should be numbered.
- It should contain ***original*** work. You are NOT allowed to copy information from the Internet and other sources directly. It also means that it is not allowed to cheat, in any interpretation of the word. You are allowed to discuss questions with class-mates but you must provide your own answers.
- Your external references for your work should be referenced in the hand in text. All external references should be complete and included in a separate section at the end of the hand in.
- The language can be Swedish or English.
- The text should be language wise correct and the examiner reserves the right to refuse to correct a hand-in that does not use a correct/readable language. Remember to spellcheck your document before you submit it.
- Write in running text (i.e. not just bullets) – but be short, concrete and to the point!
- Use a 12 point text size in a readable font.
- It's fine to draw pictures by hand and scanning them, or take a photo of a drawing and include the picture; however make sure that the quality is good enough for the picture to be clear.
- Judgment will be based on the following questions:
  - Is the answer complete? (Does it actually answer the question?)
  - Is the answer technically correct? (Is the answer feasible?)
  - Are selections and decisions motivated? (Is the answer based on facts?)
  - Are references correctly included where needed and correctly? (Not just loose facts?)

Total points: 25	
Grade	Required points
5	22p
4	18p
3	13p
U (Fail)	0-12p

Good luck! /Josef

Scenario: Variety Boggle (Boggle variants) - https://en.wikipedia.org/wiki/Boggle

Boggle is a word game played with letter dice on a grid and in which players attempt to find words in sequences of adjacent letters. Variety Boggle is a digital version containing the standard boggle-game as well as a few variants. The developer, Boog, has plans to market Variety Boggle as a handheld boggle-console. In Boog's business plan it is stated that success factors include keeping the costs for the boggle-console down and to pack as many Boggle variants into the device as possible. Boog is therefore planning on using a low performance device with as little memory as possible and wishes to structure the code in such a way that new boggle variants can be added without few to no changes of existing boggle-variant implementations.

Boog is not a very experienced programmer and has not done a very good job of designing the Variety Boggle code, even though it largely works (but with plenty of bugs) for the game variants that currently exists. Your role is to help Boog improve upon the design by following the remaining instructions in this thesis. You should also structure the design in such a way that new boggle variants can be added in the future, such as some of the variants found on https://boardgames.stackexchange.com/questions/3799/boggle-variations). (The variant Apples to Boggles is similar to the 2018 home exam: http://staff.www.ltu.se/~qwazi/d7032e2018/)

(You can assume the console is able to run whatever programming language you opt to develop the home-exam in, and that the choice of programming language itself has no impact on performance; only the quality of the code).

Rules:

1. The game requires a minimum number of two players with no maximum number of players.
2. The dice are randomised (rolled) and randomly allocated to a grid-slot on the board at the beginning of a match.
3. Only words|equations of dice that are adjoining in a 'chain' are accepted
    a. they may be adjacent horizontally, vertically, or diagonally
4. The player with the highest point total is announced as the winner at the end of each match.
5. Score is assigned as follows:
    a. Fewer than 3 dice used = 0 points
    b. 3-4 dice used = 1 point
    c. 5 dice used = 2 points
    d. 6 dice used = 3 points
    e. 7 dice used = 5 points
    f. 8 or more dice used = 11 points
6. The "Qu" dice-face is counted as two letters when calculating the score.
7. The same dice in the same order may only be used once per player (or once in total for Battle Boggle)
8. A scrabble dictionary should be used for the selected language in game modes using lettered dice.
9. For game modes using lettered dice it should be possible to receive a list of all possible words at the end of a match.
10. For game modes using lettered dice it should be possible to choose between a 4x4 grid and a 5x5 grid
11. Supported languages have their own dice configurations

Standard Boggle

12. A dice may only be used once as part of the adjoining 'chain'.

Generous Boggle

13. A dice may be used multiple times as part of the adjoining 'chain'.
14. Generous Boggle may be applied to all other boggle game-modes.

Battle Boggle

15. A word may only be used once and the point is awarded the first player that submits the word.
16. Players are notified with the accepted words submitted by other players.

Foggle Boggle

17. Mathematical equations are formed and evaluated instead of words, and dice have numbers instead of letters.
    a. Valid math-symbols are '+', '-', '*', '/', and '='.
    b. The expression left of the '=' symbol must equal the expression to the right for the expression to be valid.

Future modifications to the game

18. New game-modes should be possible to add without modifying the implementation of current game-modes and without compromising best practices, code-reuse, and good architecture design. (Example of potential future game-modes can be found on https://boardgames.stackexchange.com/questions/3799/boggle-variations).

Additional requirements (in addition to rules 1-18)

19. It should be easy to modify and extend your software (*modifiability* and *extensibility* quality attributes). It is to support future modifications as the ones proposed in the "future modifications of the game" section (you don't need to implement these unless you want to, just structure the architecture so it is easy to do in the future).
20. It should be easy to test, and to some extent debug, your software (*testability* quality attribute). This is to be able to test the game rules as well as different phases of the game.
21. Cost of the boggle-console should be kept down by improving the *performance* of the code, both in terms of CPU-use and memory use.

Questions (page 3/4)

1. Unit testing (2p, max 1 page)

Which requirement(s) (rules and requirements 1 – 18 on previous page) is/are currently not being fulfilled by the code (refer to the requirement number in your answer)? For each of the requirements that are not fulfilled answer:

- If it is possible to test the requirement using JUnit without modifying the existing code, write what the JUnit assert (or appropriate code for testing it) for it would be.
- If it is not possible to test the requirement using JUnit without modifying the existing code, motivate why it is not.

2. Quality attributes, requirements and testability (1p, max 1 paragraph)

Why are requirements 19-21 on the previous page poorly written (hint: see the title of this question)? Motivate your answer and what consequences these poorly written requirements will have on development.

3. Software Architecture and code review (3p, max 1 page)

Reflect on and explain why the current code and design is bad from:

- an Extensibility quality attribute standpoint
- a Modifiability quality attribute standpoint
- a Testability quality attribute standpoint

Use terminologies from quality attributes: coupling, cohesion, sufficiency, completeness, primitiveness - https://atomicobject.com/resources/oo-programming/oo-quality).

4. Software Architecture design and refactoring (6p, max 2 pages excluding diagrams)

Consider the requirements (rules and requirements 1 – 21 on previous page) and the existing implementation. Update / redesign the software architecture design for the application. The documentation should be sufficient for a developer to understand how to develop the code, know where functionalities are to be located, understand interfaces, understand relations between classes and modules, and understand communication flow. Use good software architecture design and coding best practices (keeping in mind the quality attributes: coupling, cohesion, sufficiency, completeness, primitiveness - https://atomicobject.com/resources/oo-programming/oo-quality). Also reflect on and motivate:

- how you are addressing the quality attribute requirements (in requirements 19 – 21 on previous page). What design choices did you make specifically to meet these quality attribute requirements?
- the use of design-patterns, if any, in your design. What purpose do these serve and how do they improve your design?

5. Quality Attributes, Design Patterns, Documentation, Re-engineering, Testing (13p)

Refactor the code so that it matches the design in question 4 (you may want to iterate question 4 once you have completed your refactoring to make sure the design documentation is up to date). The refactored code should adhere to the requirement (rules and requirements 1 – 21 on previous page). Things that are likely to change in the future, divided into quality attributes, are:

- Extensibility: Additional game-modes, such as those described in the "Future modifications to the game" may be introduced in the future.
- Modifiability: The way network functionality is currently handled may be changed in the future. Network features in the future may be designed to make the server-client solution more flexible and robust, as well as easier to understand and work with.
- Testability: In the future when changes are made to both implementation, game rules, and game modes of the game, it is important to have established a test suite and perhaps even coding guidelines to make sure that future changes can be properly tested.

Please help Boog by re-engineering the code and create better code, which is easier to understand. There is no documentation other than the comments made inside the code and the requirements specified in this Home Exam on page 2.

The code and an English scrabble dictionary are available at: http://staff.www.ltu.se/~qwazi/d7032e2020/

The source code is provided in two files, `VarietyBoggle.java` which contains the server and the code for one player, as well as all the game-states and game logic. (Run with: `java VarietyBoggle`). This must be started, and one of the game-modes need to be launched from the menu, before any of the clients are started. The client is located in `VarietyBoggleClient.java` and can be run as a person controlled online client (but current code only connects to localhost). (Run with: `java VarietyBoggleClient`). The server waits for the online clients to connect before starting the game, and the number of players can be set from the menu.

In the re-engineering of the code the server does not need to host a player and does not need to launch functionality for this. It is ok to distribute such functionalities to other classes or even to the online Clients. The essential part is that the general functionality remains the same.

Add unit-tests, which verifies that the game runs correctly (it should result in a pass or fail output), where appropriate. It is enough to create unit-tests for requirements (rules and requirements 1 – 18 on page 2). The syntax for running the unit-test should also be clearly documented. Note that the implementation of the unit-tests should not interfere with the rest of the design.

Examination criteria - Your code will be judged on the following criteria:

- Whether it is easy for a developer to customise the game mechanics and modes of the game.
- How easy it is for a developer to understand your code by giving your files/code a quick glance.
- To what extent the coding best-practices have been utilised.
- Whether you have paid attention to the quality metrics when you re-engineered the code.
- Whether you have used appropriate design-patterns in your design.
- Whether you have used appropriate variable/function names.
- To what extent you have managed to clean up the messy code.
- Whether program uses appropriate error handling and error reporting.
- Whether the code is appropriately documented.
- Whether you have created the appropriate unit-tests.

If you are unfamiliar with Java you may re-engineer the code in another structured programming language. However, instructions need to be provided on how to compile and run the code on either a Windows or MacOS machine (including where to find the appropriate compiler if not provided by the OS by default).

It is not essential that the visual output when you run the program looks exactly the same. It is therefore ok to change how things are being printed etc.