Project 1
Report
INF4121/3121

Sebasting Søberg(INF4121) & Thomas Oddsund(INF3121)

March 8, 2016

1 *Requirement 1* - Description, analysis and test cases

1.1 Description

We are going to review the program Hangman, which derives its name from the game, written in the language Java. The program consists of 5 source files, namely;

- Command.java
- FileReadWriter.java
- Game.java
- HangmanTets.java
- Players.java

Command.java only contains an enumerated type, which contains the possible commands for the program. **FileReadWriter.java** contains the logic for reading from and writing player objects to file for the scoreboard functionality, as well as a method for sorting and printing the scoreboard. **Game.java** contains the logic for the game itself, which means handling out, user input, checking input and printing the game dialogue. **HangmanTets.java** only initializes and starts the Game.java logic. **Players.java** contains the data structure for a player, as well as methods for fetching name and score.

1.2 Analysis of the testable parts of the program

As this program shipped with no requirment, we had to rely on experience and knowledge about the game of hangman to create a model of the flow of the program. This model was based both on our experience, as well as observed behaviour of the program. From this model, we could use black-box technique to design test cases for the program. The test cases were then built as pr. definition, with pre- and post-conditions, inital state, result and steps.

Out previous experience and knowledge helped us in setting up the requirements and model, from which we derived the different test cases. For the test cases, it especially made us set up cases for various forms of input, both valid and invalid, as this is the main vector from which users interact with the system.

1.3 Test cases

1. **Start Hangman**
   Initial state: No program running.
   Steps:

   1. Run "java -cp . hangman.HangmanTets" from the folder containing the hangman folder with binaries

   Expected results: Game displays welcome-statement:

   ```
   Welcome to the Hangman game. Please, try to guess my secret word.
   Use 'TOP' to view the top scoreboard, 'RESTART' to start a new game,
   'HELP' to cheat and 'EXIT' to quit the game.
   The secret word is: *One-or-more underlines*
   Enter your guess(1 letter allowed):
   ```

   Post condition: Game awaits input

2. **Correct letter entered**
   Initial State: Multiple underlines are displayed in the secret word, more then one unique letter missing.
   Steps:

   1. User enters a valid letter X.

   Expected results: Game displayes the following message;

   ```
   Good job! You revealed 1 letter(s).
   The secret word is: *Current state of secret word, letter X revealed*
   Enter your guess(1 letter allowed):
   ```

   Post condition: One or more underlines are replaced with the letter in the correct position, game awaits input.

3. **Wrong letter entered**
   Initial State: One or more underlines are displayed in the secret word.
   Steps:

   1. User enters an invalid letter X.

   Expected results: Game displays the following error message:

   ```
   Sorry! There are no unrevealed letters 'X'.
   The secret word is: *Current state of secret word*
   Enter your guess(1 letter allowed):
   ```

   Post condition: The state of the secret word remains unchanged, mistake counter increased by one.

4. **Correct letter reveals last letter(no help used)**
   Initial State: One unique letter missing and no help used.
   Steps:

   1. User enters a valid letter X

   Expected results: Game displays the following success-message;

   ```
   Good job! You revealed 1 letter(s).
   You won with Y mistake(s).
   The secret word is: *SECRET WORD*
   Please enter your name for the top scoreboard:
   ```

Where Y is the number of unique incorrect letters typed in by the user.
Post condition: Game ready for input for scoreboard

5. **Correct number of mistakes(help/no help irellevant)**
   Initial State: One unique letter missing.
   Steps:

   1. User enters a valid letter X.

   Expected results: Game displays the following success-message;

   ```
   Good job! You revealed 1 letter(s).
   You won with Y mistake(s).
   The secret word is:  *SECRET WORD*
   ```

   Where Y is the **correct** number of unique incorrect letters typed in by the user.
   Post condition: Game ready for input for scoreboard.

6. **Single non-alphabetic character**
   Initial State: Game running, awaiting input.
   Steps:

   1. Users enters a non-alphabetic character.

   Expected results: Game ignores input, asks for new character.
   Post condition: Game awaits input.

7. **Multiple characters entered**
   Initial State: Game running, awaiting input.
   Steps:

   1. User enters multiple characters.

   Expected results: Game ignores input, asks for a new character.
   Post condition: Game awaits input.

8. **Help command**
   Initial State: Game running and awaiting user input.
   Steps:

   1. User enters "help"

   Expected results: One letter is revealed in the secret word. help flag set for user in this game.
   Post condition: Help flag is set on user, game awaits user input.

9. **Correct letter reveals last letter(help used)**
   Initial State: One unique letter missing and help used.
   Steps:

   1. User enters a valid letter X.

   Expected results:

   ```
   Good job! You revealed 1 letter(s).
   You won with Y mistake(s). but you have cheated. You are not allowed to
   enter into the scoreboard.
   The secret word is:  *SECRET WORD*
   ```

Where Y is the number of unique incorrect letter typed in by the user.
Post condition: Game restarts, then displays welcome statement and awaits input.

10. **Restart command**
    Initial State: Game running and awaiting user input.
    Steps:

    1. User enters "restart"

    Expected results: The game starts a new game.
    Post condition: New game with a new word running, game awaits user input.

11. **Exit command**
    Initial State: Game running and awaiting user input.
    Steps:

    1. User enters "exit"

    Expected results: Game exits
    Post condition: No game running

12. **User enters name for scoreboard**
    Initial State: User have guessed the correct word without the help command.
    Steps:

    1. User guesses the correct word
    2. User inputs name for scoreboard

    Expected results: Name+score stored in records, new game started
    Post condition: Scoreboard has one new entry containing the playername + score, and a new game has started.

13. **Top command**
    Initial State: Game running and awaiting user input.
    Steps:

    1. User enters "top"

    Expected results: Game displays the scoreboard and starts a new game.
    Post condition: Game awaits user input.

Non-functional testing After some discussion we realized that some factors in non-functional testing are of importance. Namely:

1. **Performance** The performance of the program can be tested. Since this is supposed to be a rather simple program, a few things to test can be that it handles input fast enough(i.e. that it prints out correct/incorrect, and that it's ready for new input within a set time) and that it uses minimal amount of memory(i.e. locked device where it runs in a continuous loop can lead to a stack overflow if it continuously instantiates new classes in the wrong way).
2. **Maintainability** All systems, even a small and easy one, should be properly documented, via comments, names of variables, method names etc. It probably isn't necesearry with a huge document base for a project of this size, but for maintainability in the future, the program should have the proper amount of comments, and proper names on variables, methods etc.

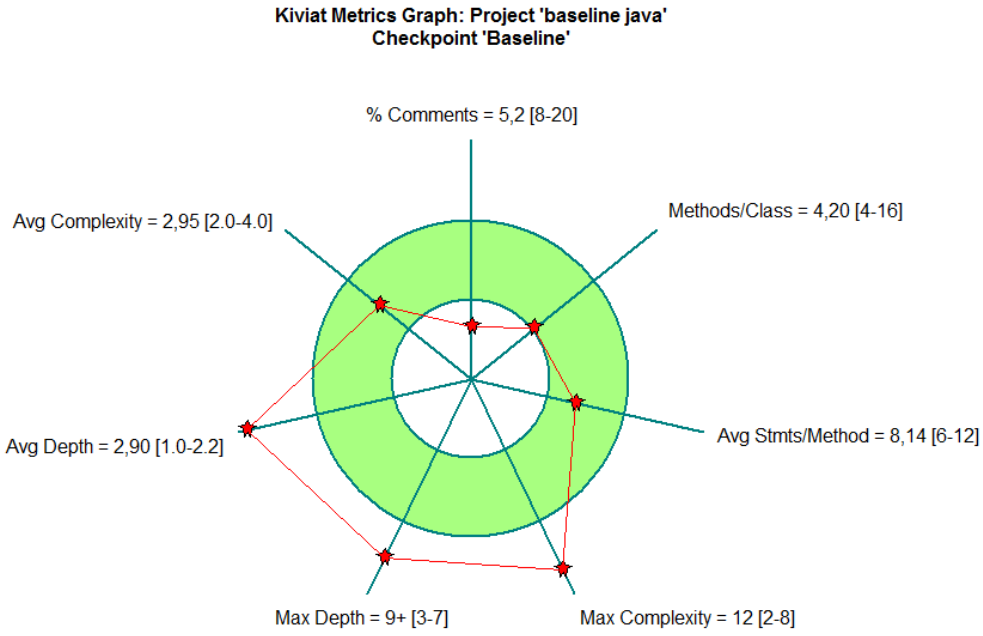
2 *Requirement 2* - Metrics at project and file level

2.1 Metrics at project level

1. The metrics of Checkpoint summary:

Parameter	Value
Files	5
Lines	441
Statements	222
Percent Branch Statements	18.5
Method Call Statements	147
Percent Lines with Comments	5.2
Classes and Interfaces	5
Methods per Class	4,20
Average Statements per Method	8,14
Name of Most Complex Method	Game.findLetterAndPrintlt()
Maximum Complexity	12
Maximum Block Depth	9+
Average Block Depth	2,90
Average Complexity	2,95
Line Number of Most Complex Method	undefined
Line Number of Deepest Block	undefined

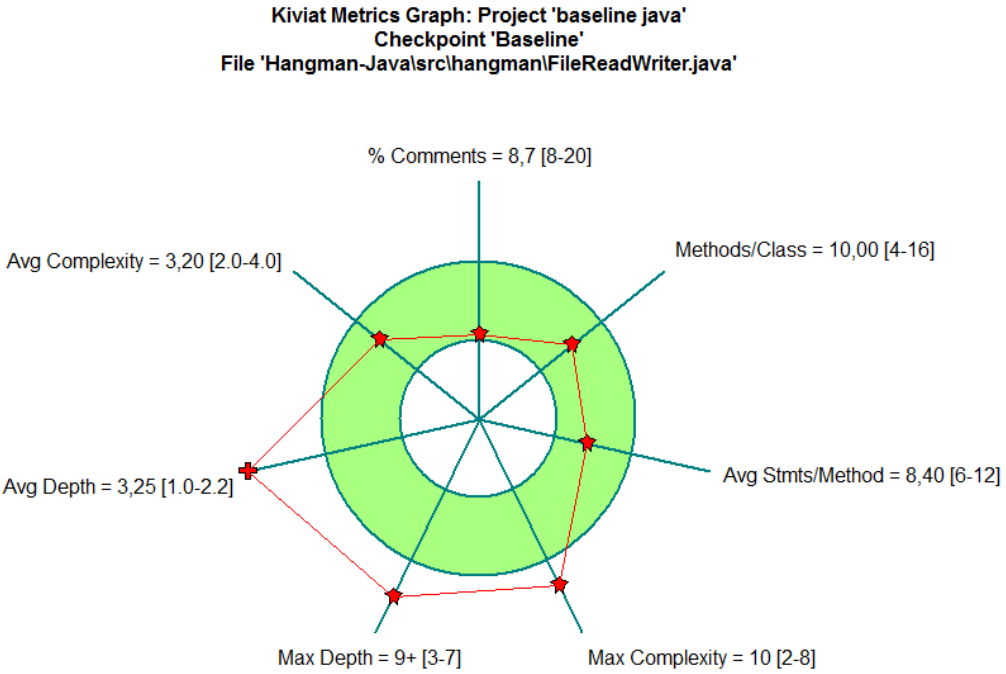
Kiviat Graph:



We think that *Comments, Max Complexity, Max Depth and Avg Depth* need to change.

2. The biggest file by the number of lines is fileReadWriter.java with 219 lines.
3. The file with the most branches is fileReadWriter.java where the branches stands for 20,8
4. The file with the most complex code is Game.java. The metrics used for this conclusion is Max complexity and Game.java's score here was 12.

2.2 Metrics at file level



1. How do you interpret the metrics applied on your file? How are they different the metrics you obtained on the whole project, compared with the metrics on this file?
   We chosed FileReadWriter.java We interpret FileReadWriter as less complex and with lower depth than the whole project. Also the comments on this file looks greater than in the entire project. The balance of the "green values" are also more balanced for Avg complexity and Methods/class than in the project.
2. Would you refactor (re-write) any of the methods you have in this file?

   Most definitly:

   1. closeFileFromReading() because it only calls another method, namely: tryCloseFileFromReading so there is no need for closeFileFromReading.
   2. nop() this method does nothing productive. Its not called from any other method in the class so it isn't used. And it is not possible to use with a reference since it is private. It only calls System.out.println(true); and outputs that 15 times. So we can erase this one as well.
   3. oldReadRecords() calls readRecords() 5 times. But for no apperent reason. This is undefined behavior because it doesnt check of many records that eventually would've been stored. It has the same problem as nop(), it is private and nothing else in this class is using it so it is rubbish.

3 *Requirement 3* -Improvements based on metrics

3.1 Metrics at project level that needs improvment

3.2 List of improved/refactored code

3.3 New metrics at project level vs old

3.4 New metrics at file level vs old

4 Conclusion

make a couple of remarks about how easy or not it was for you to maintain the code (to modify it in order to improve it). Is there anything that you would have done differently on the ini'al code to make its maintenance easier?