Kristianstad University Sweden

Kristianstad University
SE-291 88 Kristianstad
Sweden
+46 44 250 30 00
www.hkr.se

Academic report

Sustainable programming through good and clean code

Sandra Kaljula

Introduction

There is so much more to writing code than just making it do what it is intended for. The second most important things are often forgotten about like the quality and the presentation of the code. This academic report delves into the depths of good and clean code and explains the overall properties of sustainable development. In addition to that outcomes of a real-life group project is analyzed.

TABLE OF CONTENT

Method

This report is based on literature studies on good code, unittesting and test-driven development. It analyzes the observations and outcomes of the course’s assignment A02 along with generated raw data on the code size, tests and software quality metrics using static code analysis.

Results

First let start off with explaining the topic of the report. Sustainable programming is a programming approach for more sustainable development. It goes hand in hand with good and clean code, which must meet specific standards described in this report.

Good and clean code

The concepts of good and clean code do not have a specific definition, however there is a certain understanding amongst programmers on defining good and clean code. According to a survey with 45 programmers, good code was functionable, readable, testable, documented and easy to maintain[1].

Developers’ environment

For starters to write good and clean code a quality development environment is a must. The fewer distractions and more automated tools there are, the easier for the programmer to make less mistakes and focus on writing flawless code. Most of the code editors include building, debugging and executing tools[2].

Another aspect worth mentioning are the chosen programming approaches. By choosing the functional programming approach together with the automated test-driven development the programmer is more and confident and productive in delivering good and clean code[3].

Functional programming

Functional programming is a programming paradigm of declarative approach. It focuses on the result rather than how the task is solved. There are expressions called streams and lambdas that are its main properties. Selecting functional programming over the imperative approach helps to drastically shorten the code and increase the readability when transformed to the functional programming style[3]. Both of which mentioned previously are indications of good and clean code.

Test Driven Development

Test Driven development(TDD) in turn is a programming style where the tests for the units called unittests are written before the actual code. The code is developed with the help of seeing the tests fail and writing the bare minimum to pass the tests[3]. When done right the code produced through a test-driven development method keeps the code functional, minimalistic and overall easy to read. There is more thought put in the coding process which results in higher quality code and less technical debt.

Unittesting

Unittesting is done by the developer during coding. The code is divided into units which are tested for the correct results. The unit tests help the programmer or even the next team working on it to immediately notice if there is something that is incorrect in the code and helps with the debugging process. There is also a chance for false negatives and positives though. There can be blind spots[3]. For example, when changing the code new tests are needed. Which in turn can be time consuming on more complex projects making them hard to maintain. Unit tests can prevent most code smells and technical debt[3].

Software philosophies

Such fundamental solutions to common problems in software development could be called software philosophies or principles. Firstly DRY – which means do not repeat yourself, secondly KISS - to keep it simple, stupid and lastly YAGNI – you are not going to need it[5]. It could be applied in not just code but also documentation as mentioned previously.

Software documentation

Another important factor in sustainable development is continuous software documentation. It is mainly used to illustrate how the program operates. It shows the architecture as a whole and gives the reader a better understanding. One part of it is docstrings. Docstrings are like comments, they are written during development and shortly state the intentions of the files, classes and methods[4]. Another thing that should not be forgotten is semantic versioning. Semantic versioning is versioning of software during development, depending how drastic the changes where the according number in the versioning will be increased.

Static code analysis

Software quality metrics measure the quality of code. Good metrics can complement the code, when presenting it. However, they are mostly used in the development process. The programmer gets an overview of the risky spots in the code and can work on them further[6]. The metrics are generated and displayed as badges, which are usually illustrated with the grade, percentage and color.

The following metrics were retrieved from A02 with the radon tool. Appendix A shows the average cyclomatic complexity of A (1.4848). Looking over to appendix B, which displays the maintenance scores by classes. Although most of the scores are As, some of them have a score under 100. Appendix C shows the size, effort, time, bugs and etcetera for three classes.

The cohesion score retrieved with the cohesion tool. The score for the shell class was 82% and for the game class 32%. The test coverage of the project was 100% when running the coverage commands.

Discussion

Good and clean code

As gathered from the survey mentioned previously, code must meet a plenty of requirements in order to be classified as good and clean. The properties of good and clean code can be clarified by the following statements.

- The code does what it is intended to do, nothing more.
- The code does not contain obvious defects and bugs.
- The code must be easy to read and understand, so that any other programmer could start working on it without any difficulties.
- The code must be easy to maintain and extend if needed.
- The code should also not use too many resources, which in turn could make the program slow and expensive[1].

In order to facilitate meeting all these requirements, the programmer must learn about all the approaches discussed in this report.

Development environment

The first step factor in writing good quality code is the working environment, which affects the programming and debugging processes. Together with the existing tools in the market the processes can be made more efficient. An example of that can simply be accomplished by automating the whole process of running linters and unittests though using a makefile.

A makefile can hold a bunch of different commands which all can be run by just writing one word in the console. A command in the makefile can gather several external tools for linters, style mistakes and mess error detection as one command.

The results of running these tools are often shown as percentages and diagrams, which give an excellent overview on the whole project. In case of faulty code, the faults will directly be displayed, making tracking down the faults quicker and overall easier for the programmer.

By applying the automations above a significant amount of time and cost will be spared. Supposing that the developer will have more time for perfecting their code, the code will turn out better and cleaner[5].

Test Driven Development

There are many benefits to using the test-driven development approach, although it might be difficult and confusing from the beginning. However, it is a key to successful low cost and technical debt-free code. It relates back to the concept of good and clean code once again, since the code is being tested for faults and overall kept as minimalistic as possible.

From personal experience it took running some tests before getting the hang of it. After that I was aware of the goal and working methods, which made the testing process go seamlessly.

Unittesting

As a result of running unittests the faults could be spotted immediately. Otherwise, there would often be no warnings and the debugging process would be hindered. An example of that is a testcase that is expected to return a number as an integer and instead returns a number as a string. In case of using a unittests this kind of problem would be spotted right away.

In personal experience, unittests improved the quality of my code, since I only settled with the fewest lines of code that was required to solve the task. Finding faults as mentioned earlier was also simplified. Thanks to these factors I was one step forward on writing good and clean code.

Software philosophies

Although using the software development philosophies was not directly discussed in the group, the main three software philosophies were used – DRY, KISS and YAGNI. The use of these philosophies kept the code simple and repetition free, without any unneeded features. There is another principle that I personally think that should have been used - POGE - the principle of good enough. The group was always desperately thinking of the extra features that could be added. One of features that the group decided to settle with being a method that randomizes the turn. Even though A02 did not require this feature, it was still decided to be included.

Software documentation

Documentation can be quickly generated using automated tools. All it takes are existing code and docstrings that are written throughout the entire development process.

Docstrings are helpful in many ways including maintenance and extending. Furthermore, code can be found more easily by reading docstrings written in the human language. Due to that docstrings should be written with having the KISS philosophy in mind.

Another part of documentation is semantic versioning. Versioning is constantly applied in the process of development. It can be helpful to switch between different versions of the project. For example, in case of a mysterious error it might be a good idea to work back and forth with the old code and see what had caused it. Proper semantic versioning and commit messages are then essential when debugging. By having strictly stated the changes in commits and different versions to work with, the flaw can be fixed seamlessly[3].

A02 was documented by generating documentation directly from the code using pdoc and pyreverse. I personally used the UML diagrams, when opposing the other groups' projects. The diagrams gave me an excellent overview and a deeper understanding of cohesion and coupling in their program. But it is not just that, they are also useful once extending the project. By being aware of the dependencies between classes the programmer will know what else might be affected by the changes.

To avoid code smells in A02, pylint and flake8 style guides were used. See appendix D for a result from running flake8 and pylint on the finished project. At the time of submission, the group was aware of the remaining pylint errors, but did not quite understand the importance of fixing them. For that reason, it was decided to leave these errors as they are as displayed in appendix D.

Although there might be a case where the linter might be wrong. For the dice class a warning of too few methods in a class was raised. Considering the philosophy of YAGNI, there was no purpose for adding methods or completely disposing of that necessary class.

Static code analysis on A02

The average cyclomatic complexity for the project was rated A (1.4848). See appendix A. Considering that the best score is 1 and the worst is 10, the code is rather simple than complex – requiring fewer tests for full code coverage and is more maintainable[6].

Although most of the scores for maintenance seem to have a score of As, some have a score under 100, which could mean that some parts in them are slightly harder to maintain. The good maintainability score is between 20 to meaning, that the code is relatively easy to maintain[6].

Judging by the effort, volume and length of the three chosen classes in appendix C it is noted that the bigger and longer the class is the more effort and time it requires for executing it. Even though the number of bugs is low throughout the project, there is still a slightly higher chance of bugs in the longer classes compared to the shorter classes. This implies that the shorter and easier the classes are the more sustainable they are.

The cohesion score for the shell class was 82% and for the game class 32%. The score of 32% is an indication of a relatively good quality methods that work on their own. However, 82% means that there are lots of methods in the class that depend on others, which can make maintaining and extending the code a bit tricky.

All the classes except the game, shell and main classes were tested for test coverage, which made the test come back with a score of 100%. There were approximately one or two test cases per method and an assertion per test case. By not including all classes means that there could still be faults in these classes. The tested classes however had a full test coverage, which means short and simple code with a faint chance of errors.

In conclusion, the metrics were showing relatively good results. This could be a result of using the approaches mentioned in this report.

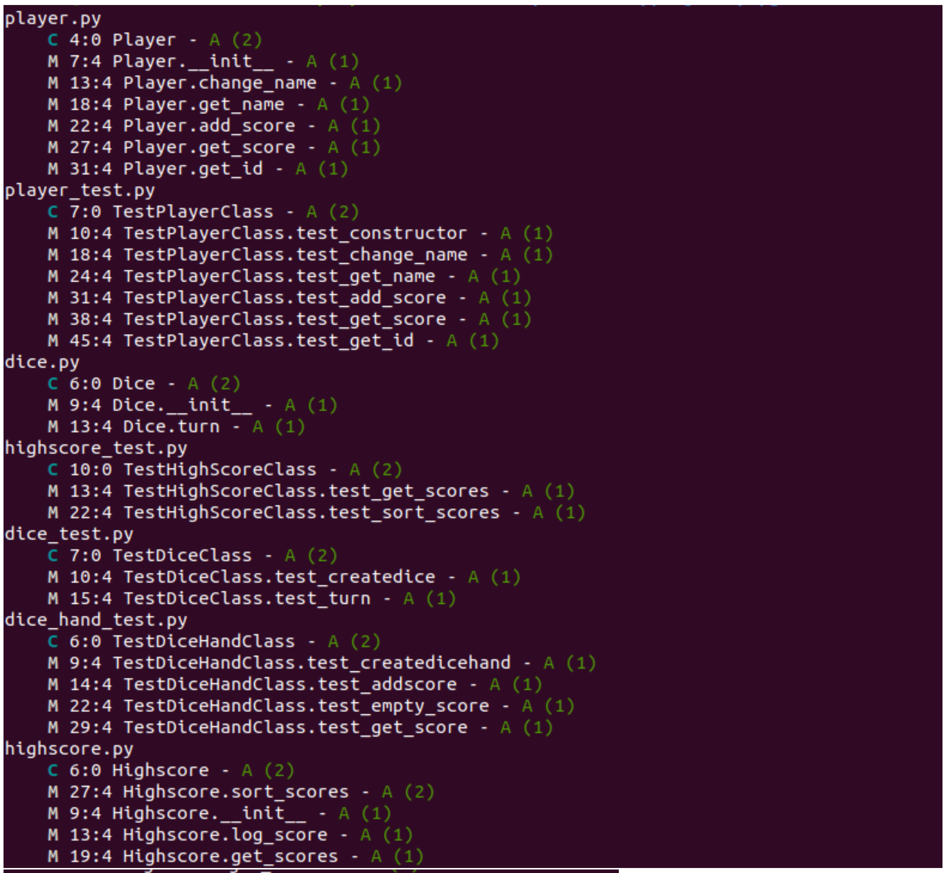
Summary

Sustainable software development is a developing style where the code matches the properties of good and clean code. The essentials for good and clean code are easy readability, maintainability and extensibility, which can be effortlessly measured with automated tools from the developing environment. The approach for coding should be of a functional programming style alongside with the test-driven development approach, whilst having at least some of the software development philosophies in mind. The software should be documented and always analyzed during the development process, which then can be generated into presentable statistics. The assignment was executed following most of the guidelines for good and clean code mentioned in this report. This could also be confirmed judging by the metrics. Meaning that the combination of the previously mentioned approaches makes the code good and clean and could be referred to as sustainably developed code.

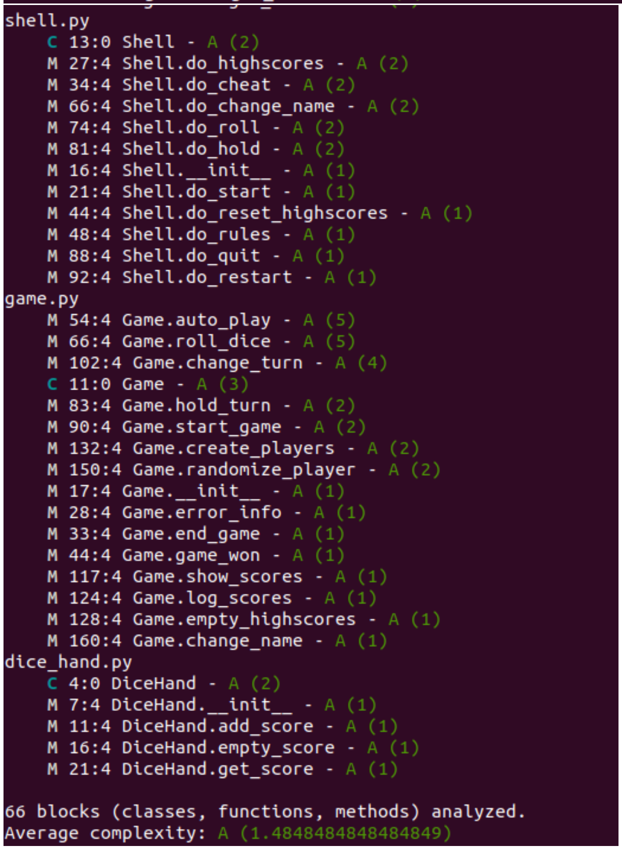
References

1. Pastrana, A. *What is good code? A scientific definition.* [Internet]. Intent HQ Engineering Blog. 2015 [updated date: 2015-03-11; cited date: 2021-03-22]. Available from: https://engineering.intenthq.com/2015/03/what-is-good-code-a-scientific-definition/
2. Technopedia. *Development Environment* [Internet]. 2021 [cited date: 2021-03-23]. Available from: https://www.techopedia.com/definition/16376/development-environment
3. Wikipedia. *Test-Driven Development* [Internet]. 2021 [updated date: 2021-03-20; cited date: 2021-03-23]. Available from: https://en.wikipedia.org/wiki/Test-driven_development
4. Reitz, K. *Documentation* [Internet]. 2011 [updated date: 2021; cited date: 2021-03-20]. Available from: https://docs.python-guide.org/writing/documentation/
5. Guigova, I. *Approaches, Styles, or Philosophies in Software Development*. [Internet]. Code Project. [updated date: 2009-03-12; cited date: 2021-03-22]. Available from: https://www.codeproject.com/Articles/33992/Approaches-Styles-or-Philosophies-in-Software-Deve
6. Microsoft. *Code metrics values*. [Internet]. 2018. [updated date: 2018-02-11; cited date: 2021-03-22]. Available from: https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2019

Appendix A

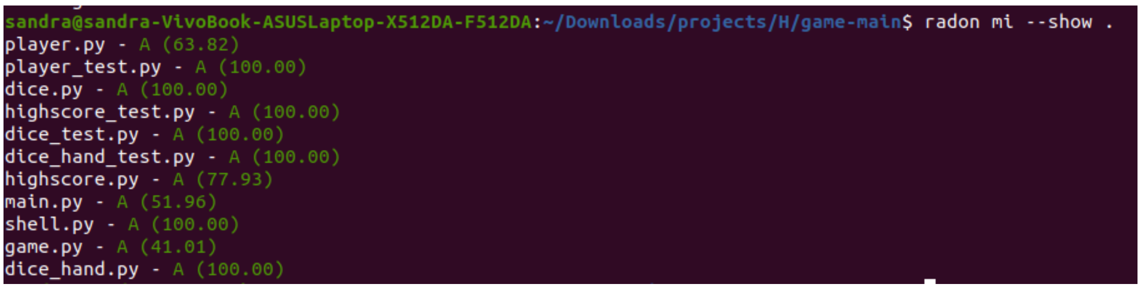


```
player.py
    C 4:0 Player - A (2)
    M 7:4 Player.__init__ - A (1)
    M 13:4 Player.change_name - A (1)
    M 18:4 Player.get_name - A (1)
    M 22:4 Player.add_score - A (1)
    M 27:4 Player.get_score - A (1)
    M 31:4 Player.get_id - A (1)
player_test.py
    C 7:0 TestPlayerClass - A (2)
    M 10:4 TestPlayerClass.test_constructor - A (1)
    M 18:4 TestPlayerClass.test_change_name - A (1)
    M 24:4 TestPlayerClass.test_get_name - A (1)
    M 31:4 TestPlayerClass.test_add_score - A (1)
    M 38:4 TestPlayerClass.test_get_score - A (1)
    M 45:4 TestPlayerClass.test_get_id - A (1)
dice.py
    C 6:0 Dice - A (2)
    M 9:4 Dice.__init__ - A (1)
    M 13:4 Dice.turn - A (1)
highscore_test.py
    C 10:0 TestHighScoreClass - A (2)
    M 13:4 TestHighScoreClass.test_get_scores - A (1)
    M 22:4 TestHighScoreClass.test_sort_scores - A (1)
dice_test.py
    C 7:0 TestDiceClass - A (2)
    M 10:4 TestDiceClass.test_createdice - A (1)
    M 15:4 TestDiceClass.test_turn - A (1)
dice_hand_test.py
    C 6:0 TestDiceHandClass - A (2)
    M 9:4 TestDiceHandClass.test_createdicehand - A (1)
    M 14:4 TestDiceHandClass.test_addscore - A (1)
    M 22:4 TestDiceHandClass.test_empty_score - A (1)
    M 29:4 TestDiceHandClass.test_get_score - A (1)
highscore.py
    C 6:0 Highscore - A (2)
    M 27:4 Highscore.sort_scores - A (2)
    M 9:4 Highscore.__init__ - A (1)
    M 13:4 Highscore.log_score - A (1)
    M 19:4 Highscore.get_scores - A (1)
```



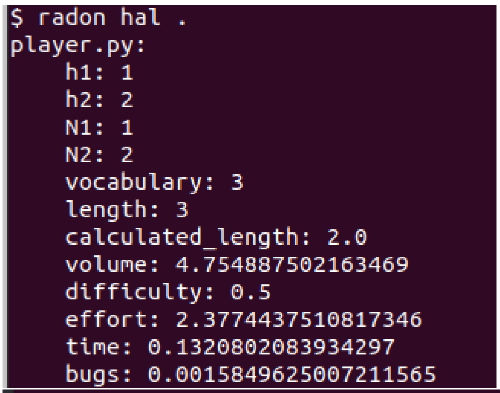
```
shell.py
    C 13:0 Shell - A (2)
    M 27:4 Shell.do_highscores - A (2)
    M 34:4 Shell.do_cheat - A (2)
    M 66:4 Shell.do_change_name - A (2)
    M 74:4 Shell.do_roll - A (2)
    M 81:4 Shell.do_hold - A (2)
    M 16:4 Shell.__init__ - A (1)
    M 21:4 Shell.do_start - A (1)
    M 44:4 Shell.do_reset_highscores - A (1)
    M 48:4 Shell.do_rules - A (1)
    M 88:4 Shell.do_quit - A (1)
    M 92:4 Shell.do_restart - A (1)
game.py
    M 54:4 Game.auto_play - A (5)
    M 66:4 Game.roll_dice - A (5)
    M 102:4 Game.change_turn - A (4)
    C 11:0 Game - A (3)
    M 83:4 Game.hold_turn - A (2)
    M 90:4 Game.start_game - A (2)
    M 132:4 Game.create_players - A (2)
    M 150:4 Game.randomize_player - A (2)
    M 17:4 Game.__init__ - A (1)
    M 28:4 Game.error_info - A (1)
    M 33:4 Game.end_game - A (1)
    M 44:4 Game.game_won - A (1)
    M 117:4 Game.show_scores - A (1)
    M 124:4 Game.log_scores - A (1)
    M 128:4 Game.empty_highscores - A (1)
    M 160:4 Game.change_name - A (1)
dice_hand.py
    C 4:0 DiceHand - A (2)
    M 7:4 DiceHand.__init__ - A (1)
    M 11:4 DiceHand.add_score - A (1)
    M 16:4 DiceHand.empty_score - A (1)
    M 21:4 DiceHand.get_score - A (1)

66 blocks (classes, functions, methods) analyzed.
Average complexity: A (1.4848484848484849)
```

Appendix B

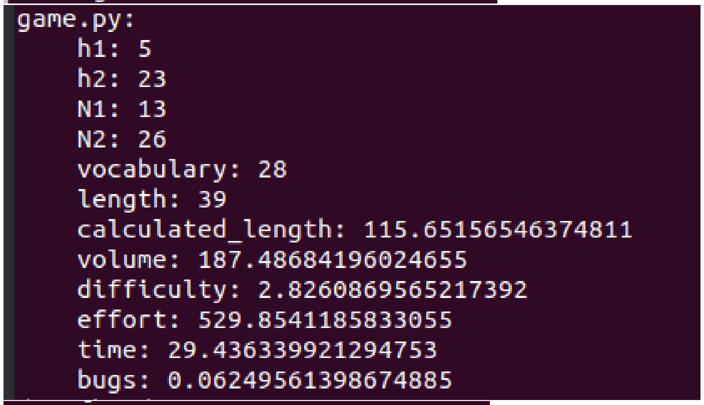


```
sandra@sandra-VivoBook-ASUSLaptop-X512DA-F512DA:~/Downloads/projects/H/game-main$ radon mi --show .
player.py - A (63.82)
player_test.py - A (100.00)
dice.py - A (100.00)
highscore_test.py - A (100.00)
dice_test.py - A (100.00)
dice_hand_test.py - A (100.00)
highscore.py - A (77.93)
main.py - A (51.96)
shell.py - A (100.00)
game.py - A (41.01)
dice_hand.py - A (100.00)
```

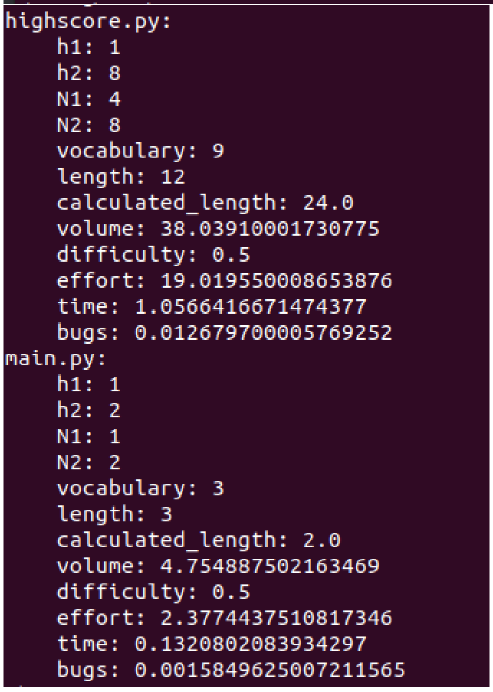
Appendix C



```
$ radon hal .
player.py:
    h1: 1
    h2: 2
    N1: 1
    N2: 2
    vocabulary: 3
    length: 3
    calculated_length: 2.0
    volume: 4.754887502163469
    difficulty: 0.5
    effort: 2.3774437510817346
    time: 0.1320802083934297
    bugs: 0.0015849625007211565
```

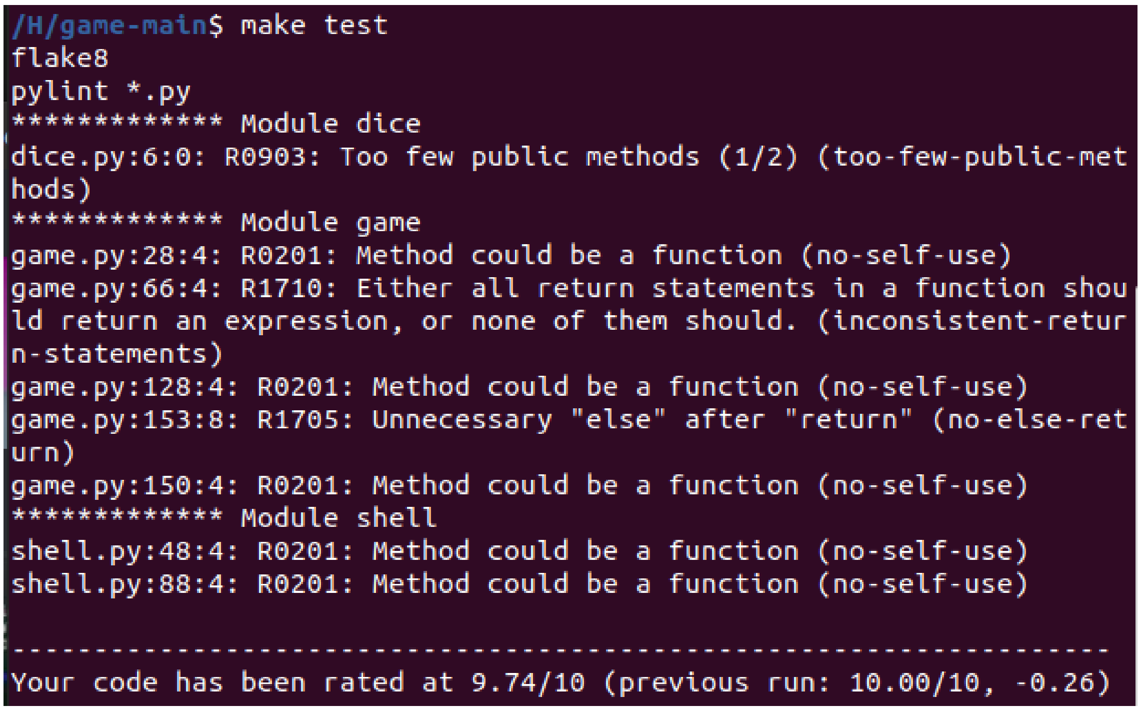


```
game.py:
    h1: 5
    h2: 23
    N1: 13
    N2: 26
    vocabulary: 28
    length: 39
    calculated_length: 115.65156546374811
    volume: 187.48684196024655
    difficulty: 2.8260869565217392
    effort: 529.8541185833055
    time: 29.436339921294753
    bugs: 0.06249561398674885
```



```
highscore.py:
    h1: 1
    h2: 8
    N1: 4
    N2: 8
    vocabulary: 9
    length: 12
    calculated_length: 24.0
    volume: 38.03910001730775
    difficulty: 0.5
    effort: 19.019550008653876
    time: 1.0566416671474377
    bugs: 0.012679700005769252
main.py:
    h1: 1
    h2: 2
    N1: 1
    N2: 2
    vocabulary: 3
    length: 3
    calculated_length: 2.0
    volume: 4.754887502163469
    difficulty: 0.5
    effort: 2.3774437510817346
    time: 0.1320802083934297
    bugs: 0.0015849625007211565
```

Appendix D



```
/H/game-main$ make test
flake8
pylint *.py
************* Module dice
dice.py:6:0: R0903: Too few public methods (1/2) (too-few-public-met
hods)
************* Module game
game.py:28:4: R0201: Method could be a function (no-self-use)
game.py:66:4: R1710: Either all return statements in a function shou
ld return an expression, or none of them should. (inconsistent-retur
n-statements)
game.py:128:4: R0201: Method could be a function (no-self-use)
game.py:153:8: R1705: Unnecessary "else" after "return" (no-else-ret
urn)
game.py:150:4: R0201: Method could be a function (no-self-use)
************* Module shell
shell.py:48:4: R0201: Method could be a function (no-self-use)
shell.py:88:4: R0201: Method could be a function (no-self-use)

------------------------------------------------------------------
Your code has been rated at 9.74/10 (previous run: 10.00/10, -0.26)
```