LO4: Limitations of the Testing Process for PizzaDronz Project

Introduction

This document describes the limitations of the testing that was carried was for the project. It also discusses suggested target coverage levels of different testing techniques used and how the actual results compare to them.

The testing carried out mainly addresses such metrics as functionality and reliability. However, many other are not covered (and not even tested for), e.g. usability, maintainability and portability.

Gaps and Omissions

Generally, I think it is worth mentioning once again that testing has been carried out only for a subset of requirements (R1, R2, R3 and R4, according to *LO1-Requirements.pdf*). Obviously, not satisfying all other requirements could in general be considered as a huge omission, but for the purpose of this limitations evaluation document, we will only consider gaps and omissions related to the actual testing that has been done.

Below is a list of all identified limitations and a suggested way of improving them.

1. Lack of documentation:
   - The tests could be well-documented (given more time-resource). This would allow for an easier testing process and its future alterations, potentially by other testers.
2. Potentially insufficient data – all synthetic data has been generated manually:
   - Some automation for synthetic data generation could be used. It would require some time to be developed but would then provide a more diverse set of possible inputs and thus a higher degree of confidence in the results of testing.
3. Lack of appropriate tools (development resource):
   - Only JUnit framework have been used to test the software. It provides a very useful functionality for unit- and system-level testing, but it is still limited, especially for mocking certain system components. Therefore, such framework as Mockito could be used to generate mock data that was necessary.
4. One of the R4 sub-criteria might not be fully satisfied:
   - This concerns the criterion which mentions that the program should robustly handle ***any*** unexpected exception during execution. As mentioned in *LO3-Testing.pdf*, all other robustness criteria can be guaranteed, but not this one. It was obviously infeasible to create a test suite which would test the application for robustness in each potential case of arising errors. It could be done (once again, given more time-resource) by automation of program runs, with various external criteria, e.g. different servers, different amounts of data, stress testing, etc.

Target Coverage/Performance Levels

Different expectations should be discussed for different testing procedures. For the 3 techniques used, we will provide a reasonable target level with a brief justification.

1. Functional Testing
   100% of the executed tests must pass. This is obvious because the program must satisfy the requirements, so (since all tests were scrutinised based on the requirements) total test satisfaction ensures the functional testing performance target is reached.

2. Structural Testing
   100% code coverage *of all components being tested* (as explained in *LO3-Testing.pdf*). This is necessary because each line of code must be executed to ensure it contains no errors. If some code is not executed, but all the tests pass, we cannot conclude the testing process has brought its desired result, and vice versa.

3. Combinatorial Testing
   100% control and data flow coverage. All possible combinations of inputs (wrong and right) must be considered. Total component code coverage does not guarantee it, e.g. there are several conditions in an *if* statement to cause certain expected behaviour and only one of them is actually covered in test suite: there will be 100% code coverage but no total branch coverage.

Testing Results Comparison with Target Levels

All three target levels specified in the section above have been reached by the testing that has been carried out. Some details on this are presented in *LO3-Testing.pdf* section *Results of Testing*.
The only one worth mentioning here is the target level for combinatorial testing. Since the testing process was narrowed down to a few requirements, it was possible to manually verify that all control flow paths have been executed. This was based on the structural testing outcomes in the sense that code coverage has shown certain lines were executed, then it was possible to construct test cases so that they cover all potential control flow paths.