# COM1028 Software Engineering

Software Engineering Report

Jamie Munro
6480591, jm01301@surrey.ac.uk

## 1. Introduction

This report provides a reflection of the design and requirements testing for my Code Snippet Manager. It contains an evaluation of each of those phases and the experience gained through the project.

## 2. Reflection on Design

The design of the project was very successful in implementing all of the mandatory functional requirements. It generates a uniform and intuitive graphical interface and includes a help menu, this makes it easy to use. All data is presented clearly to the user who is able to edit this data. All data is saved and restored across sessions.  Users can search all the snippets in the database and search terms can include text from the title, language or tags of a snippet. They can also sort their snippets or search results by date, programming language or title, and can choose whether this sorting is ascending or descending.

Originally a relational database was planned for storing the data that the program interacts with. However all the logic that the application executes on this data is performed whilst the data is held in data structures in the program memory. This means the benefits of SQL will not be effectively utilised and makes storing representations of these data structures on the disk a more efficient and attractive option. Java's built-in methods ObjectInputStream and ObjectOutputStream are used to store and load such representations. Not only does this make the code simpler and more elegant, but as both these methods are included in the standard Java library, the project does not have to include 3rd party database engine software – reducing the application's memory and disk footprint.

For the most part, integration between classes is very good, and the UML class diagram makes the relationships, associations and multiplicity constrains between classes clear. However, during the implementation of the Window class, it became much larger and more complex then originally planned. This was in part due to the large number of SWING (GUI) variables and listeners but also because many additional helper methods had to be implemented. The large number of members has made the class diagram very unwieldily. In future iterations of the project, it would be a good idea to abstract some of the logic away from the Window class to simplify the code.

Other then the UML class diagram, no other diagrams are part of the design. Originally an EER diagram was included, but as explained earlier, a relational database is no longer part of the project. There are no additional models that are not represented within the class diagram.

The search algorithm that has been implemented is probably not the most efficient way of searching the snippets. For a small number of snippets this is fine, but once a user starts searching hundreds of snippets they could begin to run into performance issues. Future iterations of the application should look at at and compare the efficiency of alternative search algorithms.

During the development of the project care was taken to insure that the system would meet non-functional requirement 2.1 portability. As the system is developed in Java, it should be able to run on any system that has a Java Virtual Machine (JVM) implementation. As the project also makes use of the SWING GUI toolkit, computer systems will need to support the SWING toolkit and provide a graphical environment. One challenge faced whilst developing the system to meet this requirement is ensuring that the database is saved in a platform-independent location. This problem was solved using the "System.getProperty('user.home');" method which returns the default file location for the user on any system (i.e. the home directory on Linux).

None of the optional requirements are implemented as they were more difficult then the mandatory requirements and ultimately could not be completed within the time constraint. In future iterations of this project, these optional requirements should be looked at again.

## 3. Reflection on Testing

Testing of the project was a combination of automated unit (Junit) tests, black box requirement testing and ad hoc testing. Ad hoc testing methods were used throughout the development of the project to ensure that new code was heading in the right direction. These ad hoc tests would involve techniques such as testing out new methods with sample data to see if they produced the expected results. Another frequently used technique was using "System.out.println()" statements to track the value of variables throughout the execution of the program. Junit tests were written for the data-orientated classes (DatabaseManager, ResultItem and Snippet) as these methods were suited for unit style tests. Due to the user-facing nature of the code in the Window class, it was not suitable for automated unit style testing and, whilst ad hoc testing was used throughout the development of the class, requirements testing was primarily relied upon.

A black box testing plan, where the tester would not require any knowledge of how the project is implemented, was created to test each functional requirement of the project. Whilst this only indirectly tests the window class, requirements testing was very successful in this role, and many issues were detected. An example of a requirements test that failed was test 19 which was testing requirement F13. To pass this test, changes to the code had to be persisted between sessions, however initially this was not the case. Requirements testing was able to pick up this issue, providing the opportunity to fix it. Whilst the underlying issue was simple - a missing call to editCode() - such an issue could have gone unnoticed if requirements testing had not been carried out.

All of the tests can be evaluated simply by visual inspection (with screenshots for evidence) and simply involve comparing the program's graphical output with the expected output. Many of the tests require the system to be in a known state, but the steps required to put the system into the required state are listed in the test plan.

Whilst requirements testing was able to assist in the fixing of many small issues, it did not lead to any elements of the project needing to be redesigned. Black box requirement testing allows developers to ensure that all of the functional requirements are met from a user's point of view, making it a very good way to test a user-centric project like this. Eventually the project passed all of the requirement tests listed in the test plan and so it can be concluded that the project meets all of its (mandatory) requirements.

## 4. References

[1] IEEE Software Engineering Standards Committee, "IEEE Std 829-2008, IEEE Standard for Software and System Test Documentation", July 18, 2008