SET09102 Coursework

40281448

**Requirement Specification**

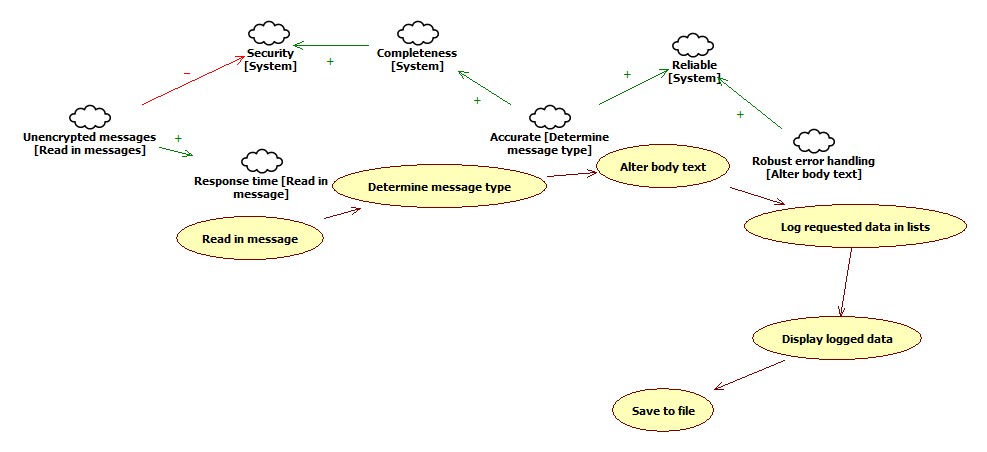


Figure NFR Use Case Softgoal Diagram

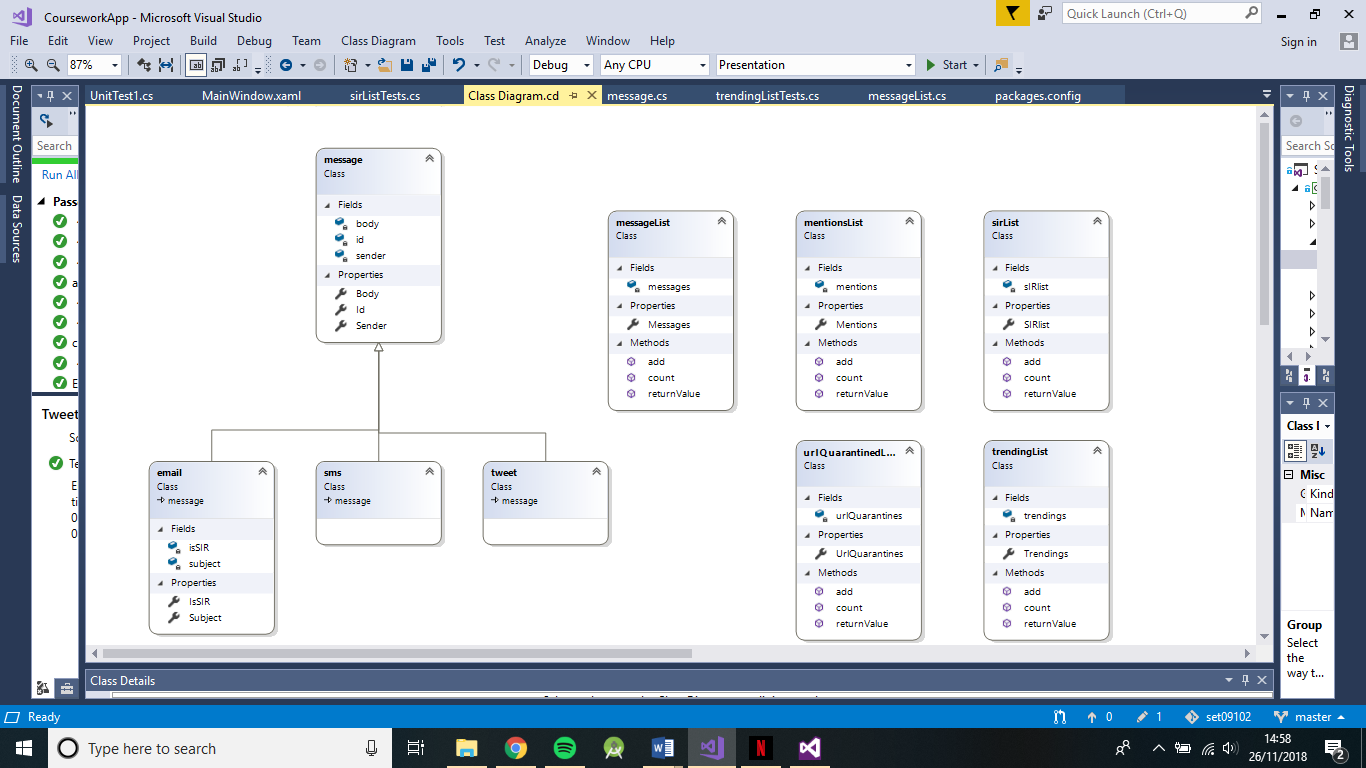
The NFR softgoals describe a system that should perform accurately, reliably, and reasonably secure. A quick response time when reading in the messages is required so as to return feedback of any errors as soon as possible to the user, with helpful error messages. The unencrypted nature of these messages aid the response time greatly, however, this leaves the messages, and therefore the information held within those messages, open to attack in multiple ways. This, without doubt, would negatively contribute to the overall security of the system, as is documented above.

The accuracy of the system in determining the type of message that is being read has an extreme impact on the overall quality of the program and as such, is crucial in delivering a complete piece of software, as requested by the client. A complete system helps to create a secure platform for the messages to be processed in.

**System Design**

The system has been designed so as to keep individual data items separate from the UI code through use of a Data and a Presentation layer. On further reflection, it would be wise to incorporate a third layer, “Business” layer. This would further separate methods and allow for easier testing of the software.

Inheritance has been used to allow the structure of the messages to have greater readability and flexibility in the code. ‘Message’ is an abstract class from which ‘Tweet’, ‘SMS’, and ‘Email’ inherit from. Although the structure of ‘Tweet’ and ‘SMS’ does not vary, separating these classes allows for a far easier implementation of conditional statements and a greater amount of readability to the code. This originally was performed through a property called ‘messageType’ however, upon refactoring this code, as an agile approach was taken, this property was dissolved and the result submitted was brought in.



The above class diagram is included within the project files of the source code for better reading, however, has been included here so as to provide quick reference to understanding assisting text.

**Testing Strategy**

The testing plan for the software will be constructed from unit tests performed at the end of the creation of classes. These tests will ensure the correct function of methods held within the classes before further testing of the overall system can be performed, to make sure the base concepts are reliable in their use. The system will be tested at the end as a whole, having a message passed in and sorted and checking this against the expected outcome.

**Test Plan**

**Objectives and Scope**

Each method will be tested in order of basic functionality, i.e. the method that requires no/the least amount of dependency on other functions to produce an output. This will ensure there is as little amount of dependant variables in play during the tests.

The scope for the message handling and processing will be far greater due to this being a test of the whole systems functionality as opposed to one methods functionality.

**Test Methods**

The unit tests will test each method individually against an expected output when run through the process. These will be run one after another to solidify the correct functionality of the classes. This style of testing will fall under the White Box testing methodology as the internal structure of the code is known and can be adapted there and then to fit the needs of the test/dictations set by the test in order to fix bugs. Extreme test cases will be used to push the limits of what can be expected in deployed use of the code.

**Environmental Needs**

The following tools outline the basis needed in order to carry out testing of the software in a suitable manner.

Visual Studio 2017 Community testing tools will be used to create the unit tests, using the ‘Assert’ keyword to compare outcomes of tests. Tests will be divided into test projects relating to the project they are testing. As such, test projects will each have a dependency on the project in question.

**Testing Schedule**

Unit tests will be created and run on completion of the class’ code before continuing on to further coding. When all tests have been completed successfully for individual classes, the system will then be tested with all components working together to produce an expected outcome.

**Risks**

Risks should be kept to a minimum when testing so as to alleviate the chances of bugs going unnoticed. The main disadvantage to the test cases is that they are written by the programmer of the software, who undoubtedly will make assumptions about the readability and functionality of the code that perhaps could hinder the uncovering of software bugs; be that logical errors in the handling of data or calculation errors when dealing with numbers and mathematics.

The use of any outside library changes adds instability to the test cases as should the library change in any way, the test may become obsolete in many ways. The use of certain keywords may change, or the syntax may require different layouts. These factors can heavily impact the future prospects of testing the code and can leave the system insecure to malicious attack due to the lack of competent testing.

**Solutions**

Had the situation differed so as to include a team of programmers, it would be wise to delegate testing to someone who had little input in the coding of the original software, to allow an outside opinion and fresh eyes to the program.

As for the risk of libraries changing, the software should be marked as tested and complete with the version numbers of all libraries at the time, with continual checks performed for updates for components used. Should a large update affect the usability of a component, an updated test should be performed to ensure the security and functionality of the software remains unhindered.

**Version Control Plan**

An online repository for code, GitHub, has been used continuously throughout the development of the software to allow a seamless and reliable integration of design iterations. As this project has been completed by one individual, the assumption will be made, for the purpose of the version control plan, that multiple team members are involved.

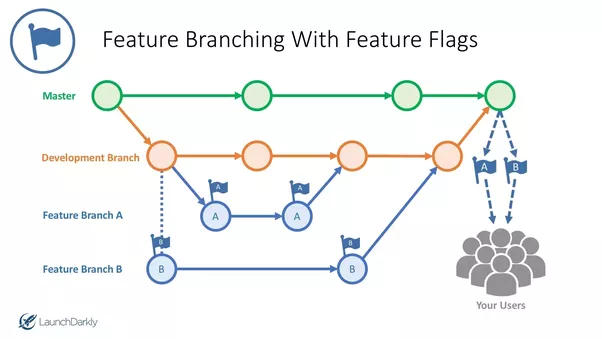
Team members will be separated by task, based on the current iteration’s requirements. The VCS will be utilised in a way that branches are created, preferably in a 3-tier manner. The team will branch from the master to create a development branch before branching from that to create their own individual branches. 

Figure 2 Branch Layout in VCS

By using this approach, the team will be able to intricately examine every increment to the app with complete control. The master branch (the live branch) is unaffected from any development work, allowing users to access the software without disruption. The development branch is taken from the master and allows the teams to communicate and build the latest increment. It acts as a hub for the teams, allowing everyone to see changes and make changes if necessary. This allows teams to be re-assigned to different tasks with ease, as the code in development is accessible by all.

From this development branch, teams create their own individual branches that are designated for the development of their task, referred to as ‘feature’ in the above figure. This keeps tasks separate from each other and allows a safe environment for teams to develop, test, and push the boundaries of code without having any effect on anything out with the current branch.

When teams complete their tasks, they are able to push their changes to the development branch, where code can be collated and reviewed before being deployed on the master branch. Before deployment, the teams can set up automated tests that run before a push is accepted in the VCS. This extra step can ensure the branches only consist of clean and usable code, with no undesirable surprises for those who may be pulling from the branch.

**Evolution Strategy**

It would be a fair assumption that before deployment of this application, legal issues with the transfer of personal information to the software carried in the messages, through whatever means, would invoke the implementation of an encryption feature. The messages would need to be securely delivered to the software and unencrypted within it, before saving the files securely – the necessity of re-encryption at this point is dependant on the manner in which they are stored, i.e. within a database that is already encrypted or other means of effective security blocking access to the sensitive information.

The evolution process would involve a mix of some corrective, adaptive and perfective maintenance, before eventually requiring at least some re-engineering to keep methods and functionality up-to-date in the processes that are used. The corrective maintenance would assess the clarity of the classes in the first regard, before expanding to ensure that the rest of the program is as easy to maintain as possible.

The system has been split into methods as far as seemed sensible to increase the readability of the software as a whole, in an attempt to follow best practice and decrease future maintenance costs. Commenting and clear indication of the purpose of methods both play imperative roles in increasing the ease of understanding of the project and allow debugging to accurately pinpoint any potential bugs or defects without complication.

Most of the maintenance cost would be during the adaptive maintenance, in which functionality would be altered to keep the software up-to-date with the current demands of the industry. Enhancements that are made after this maintenance has been performed would consume the largest part of the maintenance budget, as new requirements come into play.

# References

Justin Baker, S. (2016, August 1). *What is the best way to incorporate feature flags and toggles with version control branching?* Retrieved from Quora: https://www.quora.com/What-is-the-best-way-to-incorporate-feature-flags-and-toggles-with-version-control-branching