Edinburgh Napier UNIVERSITY

Coursework Report

Maria Luque Anguita
40280156@live.napier.ac.uk
Edinburgh Napier University - Software Engineering (SET09102)

1 Introduction

The purpose of this coursework is to develop a prototype that will validate, sanitise and categorise incoming messages in the form of SMS, emails and tweets. This report is required to detail:

1. a requirement analysis of the prototype created
2. a class diagram of the system
3. testing of the prototype
4. using version control to support development iteration and team members collaboration
5. an evolution strategy, maintainability of the system, predicted maintenance costs and evolution process and methods

2 Requirement Specification

As Steve McConnell said, the main aim of requirements gathering is helping users figure out what they want, since at the start they will not be completely sure of what the program has to do. This is why during Agile development the team maintains constant contact with the product owner and has to continuously respond to change.

There are two types of requirements: Functional (describe what the system should do) and Non-Functional (describe how the system works).

2.1 Functional requirements

Functional requirements are services that the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. It may also include what the system should not do.

These requirements should be specifically detailed or they might be interpreted in a different way depending on who is reading them (developer or user).

For the Napier Bank Message Filtering System the functional requirements are as follow:

1. Read file (.json or .txt) containing messages or manual input of messages from user
2. Validate the input and identify the type of message (SMS, Email or Tweet)
3. Sanitise the input:
   (a) detect abbreviations in texts and add their definition after them
   (b) quarantine URLs and add them to a list
   (c) add twitter mentions to a 'mentions' list
   (d) add twitter hashtags to a 'hashtag' list
   (e) identify Significant Incident Report emails and add sort code and incident to a SIR list
4. Categorise the input: SMS, Email or Tweet
5. Output the processed messages to a JSON file

These functional requirements are what then will be the use cases, which are also the user stories, even though the client will not know the technical details that each part requires.

2.2 Non Functional requirements

Non-functional requirements are constrains such as timing, development processes... These often apply to the system as a whole, not to just one feature or service.

The main non-functional requirements that the NBMFS must provide are:

1. reliability
2. efficiency
3. storage requirements must be more than enough to cope with all the data, the system must have scalability
4. security

The security requirement generates a number of related functional requirements that define system services that are required. A single non-functional requirements has the power to create many functional requirements.

These requirements may be more critical than functional requirements since if these are not met, the system could be useless. They can be divided into product requirements, which specifies the way in which the delivered product should behave, organizational, this requirements are a consequence of policies and procedures of the organisation, or external requirements, which arise from legislative requirements or interoperability requirements.

The product requirements for the NBMFS are:

1. **usability**: the program must be easy to learn and easy to train the staff that will be using the program, user friendly and should have error tolerance, the program should never crash, if any it should display the error to

tell the user that something is wrong but never crash the entire system

2. **robustness**: if there is an error that crashes the whole program, the system should not take long to restart after the failure
3. **efficiency**: the system must not occupy a lot of space and should pass the performance test
4. **dependability**: making sure the program delivers the expected outcome on time and detailed
5. **security**: the program's data should not be able to be accessible to anyone who is not authorised to see or manipulate said data

The organisational requirements includes environmental, operational and development requirements. In the NBMFS the users will only be the staff working in the organisation, and they must be able to log into the program with their organisation's ID and password or identity card, which is not related to the program itself but must be implemented and therefore is a non-functional requirement, since it accounts for how the system should work.

And finally the external requirements include requirements such as regulatory, ethical, legislative, accounting and safety/security requirements. The NBMFS should implement data privacy policy of the country it is used in.

3 System Design

3.1 Use Case Diagram

The purpose of a use case diagram in UML is to demonstrate the different ways that a user might interact with a system. The use case diagram for this system is shown in Figure 1 in the Appendix.

3.2 Class Diagram

Class diagrams represent the structure of a system by modeling its classes, attributes, operations and relationships between objects. The describe what must be present in the system. The class diagram for this system is shown in Figure 2 of the Appendix.

4 Implementation

4.1 Version Control Plan

This system is to be developed in agile approach, this means, dividing tasks into short phases of work and frequent reassessments and adaptations of plans. The Agile Manifesto specifies the four key values that software developers should use in order to guide their work. They state that individuals and interactions, working software, customer collaboration and responding to change are the most important aspects in agile development.

For this project, Agile methods would be the best since it would allow the team to deliver high-quality software quickly and continuously while promoting team member collaboration and communication with the client.

During the implementation part, version control should be used to help the software team manage changes to the source code over time, this allows the developers to go back and compare earlier versions of the code to help fix the mistakes without interrupting the rest of the team members.

The plan would be to start with an iterative development, which means using Agile Methodologies. This development breaks down the the project into smaller chunks of code that is designed, developed and tested in repeated cycles, and with each iteration, additional features can be added until the final product is delivered.

The first step is requirements gathering. In Agile development the owner describes what the final product should do using user stories, since many agile methods argue that producing a requirements document is a waste of time as requirements change all the time. In SCRUM, the user stories requirements are put on a backlog as product backlog items in an ordered list with the most important requirements at the top. The product owner is the one who orders the backlog. This backlog should be finished at the end of a sprint (2-4 weeks), and a MoSCoW Analysis should detail what features should be implemented in the next sprint backlog.

Every day there will be meetings where the progress will be discussed. This promotes collaboration amongst team members since everyone will talk about what they have done since the last meeting, what they will do for the next meeting and what is stopping them from doing it. Another way of promoting collaboration is by doing paired-programming, where 2 developers use one computer and they learn from other people's ways of working, and allows them to be more open to constructive opinions.

Version control is used during the whole process, the iterative development and the team's collaborations. There are three types of version control:

1. Local version control system: it only keeps the differences between the file's content at progressive stages and then recreates the file's contents
2. Centralised version control system: keeps the files in a common place where all the team members have access to from their local machine. Only the last version of the files are retrieved.
3. Distributed control system: stores the history of the files on each and every machine locally and syncs the local changes made by the user back to the server. This would be the best option for this system since it provides a collaborative working environment and it is easy to use and manage.

During version control, the master copy of all versions is stored in a repository, which acts as a server, while the version control tools act as the clients. To see the different versions of the NBMFS system, each commit can be accompanied by a tag, which to make it easier to understand and remember will be written as a number and a description of the changes. The version control system will also record who the author was and the changes that have been made. If any team member has any problem with another member's commit,

they can see who it was and discuss it together face-to-face, as well as go back to a past version if they decide it was better before.

After each sprint of the development iteration, other features should be implemented in the current version. Team members will be working on different parts of the system, together or alone (paired programming) but they all have to make sure that their parts work well with the current version before committing to the version control system, so that if they ever have to go back, they retrieve a program that does not crash. For this to happen, the version control system provides the branching tool. A branch is like having a totally separate repository where the developer can see the source code it contains and operate independently of the rest of the team members working on the master branch or other branches. When the feature is implemented with the current version and it works like required, then they can merge their branch with the master branch. Merging eliminates the need to cut and paste changes back.

The conflict resolution method used will be Optimistic locking, where every developer gets to edit any file, but the version control system will not allow them to check in a file that has been updated in the repository since they last checked it out. This encourages collaboration among team members because the version control system will sometimes attempt to merge both changes but it might not be the desired outcome, therefore the team members must collaborate together to produce a version that meets both developer's requirements and works well together.

5 Testing

Software testing is used to find errors in a program. It is an investigation that provides the client with information about the quality of the system and gives developers the chance to fix them and create better versions of it.

Testing requires test cases, test data and test results, which compared to test cases provide the test reports. When we run a software test, we execute it with test data and observe the system's operational behaviour.

Software inspections should be done to discover anomalies and defects, as well as considering broadening quality attributes of the program. The inspections will check conformance with a specification but not non-functional requirements.

5.1 Test plan

The objective of software testing is to find the most errors with the minimum amount of time and effort. All the tests should be traceable to customer requirements, planned long before the actual testing starts and should be conducted by someone who wasn't involved in the system's development. Testing is applied at 3 levels: Unit Testing, Integration Testing and Acceptance/Validation Testing.

In Unit Testing each module is tested individually to find bugs in logic, data and algorithms. White Box testing is used for this, where knowledge of the software's internal design is used to develop tests. The test data is derived from the component code, and then the test data is used to test the component code, which then produces the test outputs.

The objective of testing the NBMFS units is to check if exceptions are being raised if there is an error or if the value has been correctly set to what it is supposed to if the input is in a valid format.

Integration testing finds bugs in interfaces between modules. Black box testing, top-down and bottom-up are the methods used to find out these test results. Black box testing is different from white box because it requires no knowledge of the code, and focuses only on the requirements and functionalities of the system. Incremental integration tests small units and only adds them to the main program when they pass all the tests. This can be done by the top-down or bottom-up methods. Top-Down integration testing integrates modules downwards through the control hierarchy while Bottom-Up tests the lowest levels first and works upwards.

Finally, validation testing assures that the software meets all functional and performance requirements. This testing is done using the Black box testing method, a regression testing can also be done in order to see if the software still meets its requirements in light of changes to the software, or use stress and back-to-back testing. Stress testing exercises the system beyond its maximum design load and checks if the system suffers degradation as a network becomes overloaded.

Stress testing can be used to test the NBMFS to find the scope of the program and see if the objectives are being met. A text file with many, many entries will make the program have to cope with a lot of data and should not degrade in efficiency.

whitebox/blackbox/unit

objectives and scope

test items

tasks

deliverables

test schedule

5.2 Test methods

The unit tests are made by testing every single attribute of the classes and asserting the expected result with the actual result or catching the exceptions when they are supposed to appear.

Manual testing was done to make sure all the buttons in the program trigger an action and there are no program crashes.

exception testing

ui testing

5.3 Test cases

functional and non functional req

A test case consists of a description of the objective of the test, the input test data needed to conduct the test and a description of the expected result.

5.4 Test outputs

how to fix them table

possible tools

possible risks and solutions

5.5 Analysis

environmental needs

6 Evolution

Evolution, software updates, or new requirements emerge when the program starts being used. Errors are found by users that developers couldn't find; performance or reliability might need to be improved; the environment might changes or new computers might be added to the system. All of these should be maintained and should keep the program working and up-to-date.

The evolution processes depend on the type of software of the system, the development processes used and the knowledge that the current workers involved have of the system.

6.1 Evolution Strategy

Evolution or maintenance?

Maintainability of the system

Predicted maintenance costs

Evolution process and methods?

Figure 1.
Use Case

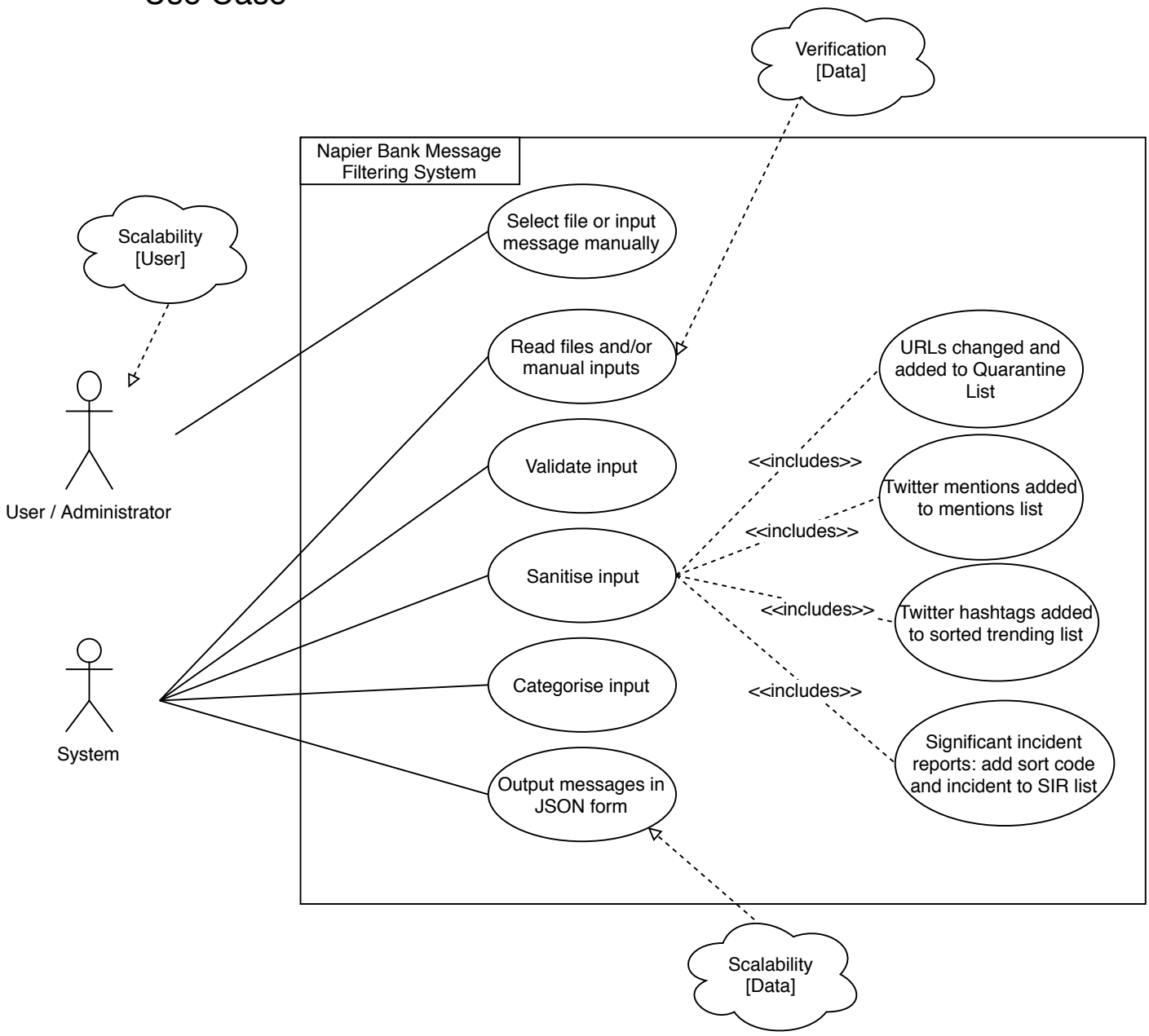

Figure 2.
Class Diagram