Software Engineering Coursework [SET09102]

Stewart J Anderson

School of Computing, Edinburgh Napier University, Merchiston Campus, Edinburgh, Scotland
40345422@live.napier.ac.uk

Abstract. In this coursework, I have explored and created a system for a fictional bank (Napier Bank) which cleans and sanitises messages / communication through its system, namely SMS, Emails and Tweets.

The bank itself is a medium-sized local bank with many thousands of users. The bank operates from one main headquarters and copious branches. The systems itself categorises these messages, processes them and proceeds to output these messages into appropriately named JSON files.

Throughout this report, I will be discussing a requirement analysis I have undertaken for the Napier Bank Message Filtering Service; a class diagram I have produced which abstracts the main functionality of the system; a test plan, accompanied by test methods, cases, outputs and analyses; a version control plan and an evolution strategy for this software.

1 Requirement Specification

Throughout a developer's lifetime in industry, they will usually encounter these two types of requirements. These are Functional Requirements (FRs), and Non-Functional Requirements (NFRs).

1.1 Functional Requirements

These are defined as services a system should be providing – or how it should be reacting to a set of inputs. For this specific system, these are as follows:

- Validate the inputs to the system
- Classify the message as a:
    - SMS
    - Tweet
    - Email
    - Significant Incident Report (SIR)
- Sanitise the validated inputs appropriately:
    - Detect abbreviations in SMS messages, then expanding these abbreviations in parentheses.
    - Quarantine URLs from Emails and SIRs.
    - Extract all "mentions" from tweets and append to a "mentions" list.
    - Extract all "hashtags" from tweets and append to a "hashtags" list.
    - Differentiate a SIR from an email:
        - Add the Sort Code and Nature of Incident to a SIR information box.
- Serialise and output these messages to appropriately named JSON files.

1.2 Non-Functional Requirements

"A non-functional requirement (NFR) is a requirement that specifies criteria that can be used to judge to operation of a system, rather than specific behaviours."[1]

To fulfil the statement above, Napier Bank Message Filtering Service must *at least* be/have:

- Durable – Over the course of the software's life, it should remain functional without the requirement of excessive maintenance.
- Efficient – Utilise a system's resources properly for a given workload. This could be through loading and saving files in example of this system.
- Extensible – Allows provisions for future growth of the software system. Can be through addition of new features or modifying existing functionality.
- Resilient – "The ability to provide and maintain an acceptable level of service in the face of faults and challenges to normal operation."[2]
- Robust – The system should be able to handle exceptions appropriately and handle erratic/erroneous output.
- The system should also implement the national data retention policies for legislative and security concerns.

Non-Functional Requirements (also known as qualities) may be deemed unequivocally more essential than Functional Requirements, as without these, the system is rendered effectively useless. These qualities can be separated into two primary categories - Execution and Evolution.

- Execution Qualities – Observable at run time (operation), these qualities include usability, security, and safety.
- Evolution Qualities – These are embedded in the structure of the system, which includes scalability, testability, extensibility, and maintainability.

2 Software System Design

2.1 Class Diagram

A class diagram's duty is to accurately represent the structure of a software system by creating a visual representation (model) of its relationships between objects (including multiplicity), operations (the actual code), attributes (properties) and the classes themselves, which can be modelled into objects. Class diagrams themselves exhibit an abstraction of what the system does (**Fig. 1**), showing the standard functionality while removing the implementation.

2.2 Use-Case Diagram

At its most basic level, a use-case diagram (**Fig. 2**) is used to convey the relationship between the user's interaction and the software system where the user is concerned. These are fantastic diagrams to use due to their ability to communicate with stakeholders, as they can be used in conjunction with additional diagrams and full documentation to give the stakeholders a complete technical and functional view of the system.[3]

3 Testing

In the development world, testing should be viewed as an investigatory process, which provides detail to the client regarding the quality of this software system, while giving the developers ample opportunity to rectify these mistakes and create superior versions of the software.

This process is all about 3 things: Test Data, Test Cases and finally, the Test Report. Each test that is ran requires sample data where we expect the system to behave in a particular way. This is then observed and noted by the individual(s) responsible.

3.1 Testing Plan

This software system (Napier Bank Message Filtering Service) was written to be a TDD (Test-Driven Development) system. Initially, copious tests were written prior to code composition, while referring continuously to the requirement specifications to ensure the source code passed the specified tests. This benefits the system vastly as it significantly reduces the debugging process and the desire for regression testing – re-running tests to ensure a previously developed and tested software still performs after a change[4] –.

My test items for the system were essentially all the classes contained in the business layer and the data layer, as they provided a ton of functionality to the system. This primarily covered the functionality of:

- Message
- SMS
- Tweet
- Email
- SignificantIncidentReport
- ServiceFacade
- LoadSingleton
- SaveSingleton

Unit testing was the optimum approach for this project, where an individual function's behaviour could be investigated against its target behaviour determining its "fitness". When a function is deemed to be fit for use, it is ready to be fully implemented to the code so the system can appropriately process parts of the message. This will reduce user input error and vastly improve maintainability.

Importantly, we can test an application at a few different levels. We can go through:

- Unit Testing
    - Each individual component (module/function/procedure/method) is thoroughly examined to find mostly logical errors.
    - To test this, we utilise the white box testing methodology.
    - We are certain about the structure of the software and use this to create our tests.

    - We then utilise testing data to check the functionality and behaviour of the component.
    - This creates a test output!
- Integration Testing
    - This is useful for finding bugs between components.
    - This is achievable through black box testing, bottom-up testing and top-down testing.
    - Amazingly, we do not require any background knowledge of the code structure, as this type of testing is focussed solely on functionalities and requirements!
    - We start off by incrementally testing very small units of code and only append these units to the software system when all the units' tests pass.
- Validation Testing
    - Simply put, validation testing compares the user input and checks to see if it matches what the system is expecting.
    - If you provide incorrect data, expect an exception being thrown.

In addition to the methods listed above, we can also stress test this system by providing a csv file which has many thousands of entries in it and seeing how the system copes with the message processing. The program's efficiency should not degrade past a certain extent.

3.2 Testing Methods

The tests created were made with consideration to the functionality and procedures that were created in each of the business classes. It was asserted in the test classes that each of the tests must return true for them to pass. There were a small number of tests that were to throw exceptions when the wrong input was identified, so it was asserted they must throw these exceptions for the tests to pass, which saves the test from failing.

An early-stage alpha manual user interface test was conducted, ensuring that any controls being used by the user were not to crash and have the program behave unexpectedly. A deployable version of the user interface was dispatched to the test subjects, where they were asked to share their screens and talk through their processes. These tests were completely unfeasible to conduct in-person, unfortunately, due to COVID-19.

3.3 Test Cases

"In software engineering, a test case is a specification of the inputs, execution conditions, testing procedure, and expected results that define a single test to be executed to achieve a particular software testing objective, such as to exercise a particular program path or to verify compliance with a specific requirement."[5]

Test cases and their outputs can be seen in **Table 1** of the appendix.
A summary of all tests passing can be seen in **Fig. 6** of the appendix.

3.4 Test Outputs

Please assume for this table, where properties or methods are being accessed, these are indeed valid values for what is trying to be achieved; and anything listed as Class#Method is utilising the correct parameters of that method. This is only due to the layout of the report that has been utilised, so a lot of text cannot be entered otherwise the table becomes messy. Please also note that single class names followed by parenthesis are constructors for classes.

3.5 Analysis

This system has been rigorously tested to ensure it is as fool-proof as it possibly can be, ensuring that regardless to what the user attempts to enter into the system, the system should respond rationally while providing the user with appropriate guidance on proper system usage.

4 Version Control Plan

When system creation commenced, it was thought it would be easier to tackle this problem with an agile approach. This allowed a focus on high-quality output while being able to promote importance on small incremental updates to the repository, so should something go wrong, the developer can always go back to what previously worked.

The fundamental place to start with this project is requirements gathering. These requirements are often dictated by the project owner via project stories, as a full-blown document is a complete waste of time, as requirements may change at the drop of a hat. These are added to a product backlog, which is usually a Kanban board.

After discovering what these requirements are from the project owner, a MoSCoW (Must-Haves, Should-Haves, Could-Haves & Would-Haves) analysis is undertaken to prioritise the features so we know what we must work on to have the core aspects of the system functioning prior to the first tests going underway.

This process was initiated with another process called functional decomposition. As the process sounds, much larger problems where broken into smaller problems and because this is a recursive process, these problems can continue to be broken down into even smaller problems. This helped in understanding the problem given initially and aided in the development process.

Since there was an initial idea on how to approach the issue at hand, a way to store it was required. It was felt it would be appropriate to explore appropriate version control methods. As it happens, there are three main types of VCS (version control systems):

- Local VCS (**Fig. 3**)
    - For simplicity, this version control system is the choice of many for its easy implementation.
    - It requires you to locally store files on your system in multiple folders with versions alongside.

    - However, it is ludicrously prone to errors as you could accidentally write to the wrong directory overwriting the data in a previous version.
- Centralised VCS (**Fig. 4**)
    - Issues arose from the previous VCS, as people needed to collaborate on projects without having to pass a hard disk around regularly.
    - To combat this, a centralised system was developed. This enabled collaboration through a single server containing all the versioned files.
- Distributed VCS (**Fig. 5**)
    - There was still a massive outstanding flaw with the previous VCS, if it was stored on a single disk and it got corrupted, you lost everything.
    - Hence why the distributed VCS was created.
    - This enables each client to "mirror" the repository, including the full history on their machine.[6]
    - Which allows every user to locally store their changes and upload their changes to the server.
    - Should any conflicts arise, you can always pull the latest changes and re-commit later.
        - This is through a process called optimistic locking – a user cannot update a file that has been updated, where the changes on the server are not on your local copy of the repository.

Obviously, the Distributed VCS was the most secure way to ensure there were no outstanding changes which would create conflicts. Handily, the software being used to create this software system (Visual Studio) has git integration, so automatically committing, pushing, pulling, merging and creating branches at will from within the software without having to leave for my command line or go to my repository online was relatively easy.

In hindsight, the best way to develop this system better would be to fully utilise the properties of git and use feature branches, which push to a development branch. When it is discovered that a development version is ready, it should then be pushed to the master branch. Eventually this leads to a full working version being placed on the release branch where it is then marked and tagged as a release through the version control system. Unit tests are placed on their own separate branch and are possibly accompanied by integration tests. Should the matter arise that an outstanding major error has occurred in the program, a hotfix branch exists to address this issue, where urgent patches are made. This is focused on by the entire development team and they cannot move on until this issue has been resolved. This process is often better known as GitFlow and aids in parallel development and team scalability.

5 Evolution

Evolution emerges the second a software system is deployed to the client, where there is a potential alteration to the system requirements, or the system is required to be updated in some manner.

The process of evolution dictates there are three core maintenance types:

- Corrective Maintenance
    - Repairing bugs, or;
    - "Identifying, isolating, and rectifying a fault so that the failed equipment, machine, or system can be restored to an operational condition."[7]
- Adaptive Maintenance
    - This is used to allow software systems to operate appropriately on other operating systems that they were not initially built to support. This could be a WPF (Windows Presentation Form) application being made available for MacOS.
    - This type of maintenance is not just for vast changes in environment e.g. operating systems, this could also be for software designed to run/operate hardware on different processors (ARM vs Intel).

- Perfective Maintenance
    - This type of maintenance is not necessarily to handle with what goes wrong upon execution, but more as an aftercare for the software system.
    - It is utilised by the development team as a measure to keep the software execution relatively efficient (if not more) as time progresses and operating systems/hardware become unsupported.
- Preventative Maintenance
    - This is used further along the lines as a bug will probably start out as a minor issue.
    - Employing this maintenance strategy will allow the developer to patch the fault before a latent fault becomes a colossal fault.
    - This is not as critical in garbage-collecting languages like C# or Java, but essential in non-memory-managed languages like C or C++, where memory leaks can occur.

Before you become aware of the financial concerns, you first must consider whether you must develop to maintain or develop to function. What is meant by this; is if the development team do not hold a contractual agreement with the client, you do not have to build for maintainability as your software system is built as a one-off. If you do hold a contract with the client, you should be intelligent about how you approach the development of the system. This is because you will be liable for fixing any issues that arise from frequent use of the system. You may also want to take into consideration how much of a budget you want to put the on the development of the software system; as the more you spend on the development process, the less you'll eventually spend on the maintenance of the system. Which is often referred to the coined term "speculate to accumulate".

5.1 Evolution Strategy

The final solution for the Napier Bank Message Filtering Service was created with a three-tier architecture in mind. The three layers are:

- Presentation Layer
    - This layer is designated to the User Interface.

    - This can either be one or multiple connecting user interfaces which the system may make use of.
- Business Layer
    - This layer is dedicated to the business classes.
    - Here, you will find all the ADTs (Abstract Data Types) for what we want to store.
    - Sometimes, you may find the developers making use of design patterns (like the Façade Design Pattern) which can often hide away implementation of business objects.
- Data Layer
    - This layer is typically where you would find the classes responsible for retrieving data and general persistence.
    - The layer itself is fundamentally crucial, as without data being inserted and extracted, any sophisticated system is generally useless.

This style of system would often imply it was written with modularity at its heart. Modular code implies that it is easier to use, while also being easier to maintain as you can modify implementation while keeping method signatures and accessibility the same.

The first and most important type of maintenance this system will require through its lifetime, is most likely corrective maintenance. This is because the client or end-users may potentially find minor issues in the system that the developer/development team have not encountered. Eventually, they would be required to employ adaptive maintenance to allow other end users on alternate operating systems access and use the software properly without having to use a virtual machine or emulator.

In the obvious eventuality the software requires maintenance, this is predicted based on a few factors:

- What will categorically be the most expensive to maintain and repair?
    - In terms of the software, because the system is hosted 100% on the user's hard drive (or a local file server) there is nothing to fret about in terms of hardware. The system is relatively lightweight. Any issues with hardware or possible data corruption are completely on the IT department.
- What could the lifetime costs of maintenance be?
    - Statistically speaking, these could range from double to almost 30x the initial development costs. This is simply down to when the system requires the maintenance, depends if the system is legacy. If it is, you then need to evaluate how common the language use is before you start employing developers to work on the software, or if you should move to another system entirely.
- What changes to the system could be expected over the next year?
    - Usually, the client never actually knows what they want. Their decision-making process is relatively volatile and as the development process continues, the requirements for the software may change erratically. When you start talking about first use after a proper release, they absolutely will have many requests

for changes, which you will probably have to comply with as it will better suit the needs of the business in question.
    - Perhaps the Message Filtering Service could sit in the cloud and monitor the messages being passed inside a business' network?
- Which parts of the software system could be affected by these change requests?
    - Primarily it is observed that the User Interface will be first affected by the changes that the client may request. The client may request different font sizes, user interface colours, or just general appearance/ accessibility features (colour blind, screen reader compatibility)

When the software grows old and maintenance costs are stacking up, you cannot possibly justify keeping up the current structure of the program. Instead you invest in a team of software engineers who can restructure the whole system, using new engineering paradigms giving you the same system, but have it working like new again. This process is called re-engineering.

In absolute worst-case, the system can be scrapped by the business as it is no longer fit-for-use. The client and team of software engineers should take careful consideration when thinking about this process. They should be considering the value of the software for the business' values.

References

1. Non-functional requirement, https://en.wikipedia.org/w/index.php?title=Non-functional_requirement, last accessed 2020/11/08
2. Resilience (network), https://en.wikipedia.org/wiki/Resilience_(network), last accessed 2020/11/08
3. Use case diagram - https://en.wikipedia.org/wiki/Use_case_diagram, last accessed 2020/11/11
4. Regression testing - https://en.wikipedia.org/wiki/Regression_testing, last accessed 2020/11/11
5. Test case - https://en.wikipedia.org/wiki/Test_case, last accessed 2020/11/12
6. Git – About Version Control, https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control, last accessed 2020/11/12
7. Corrective maintenance, https://en.wikipedia.org/wiki/Corrective_maintenance, last accessed 2020/11/13

Appendix

Fig. 1. My Class Diagram, showing a three-tier architecture

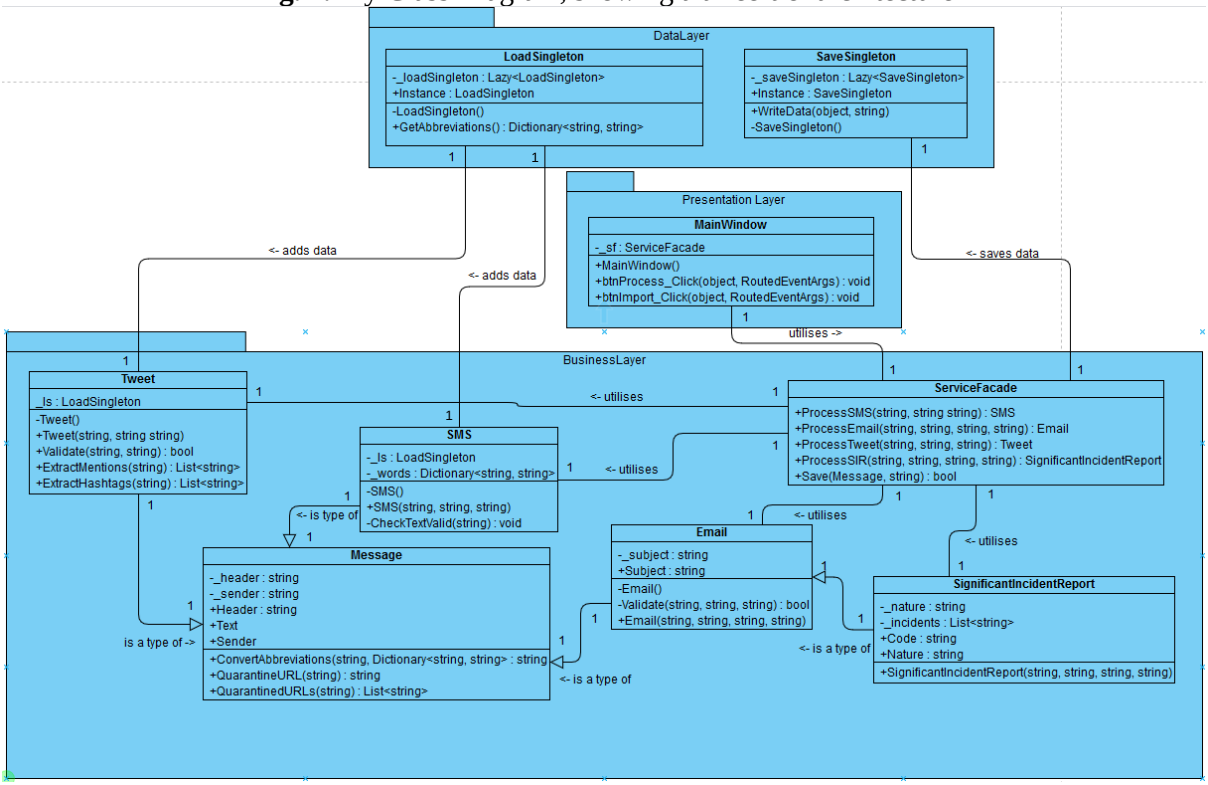

Fig. 2. My use-case diagram, showing how the user interacts with the system.

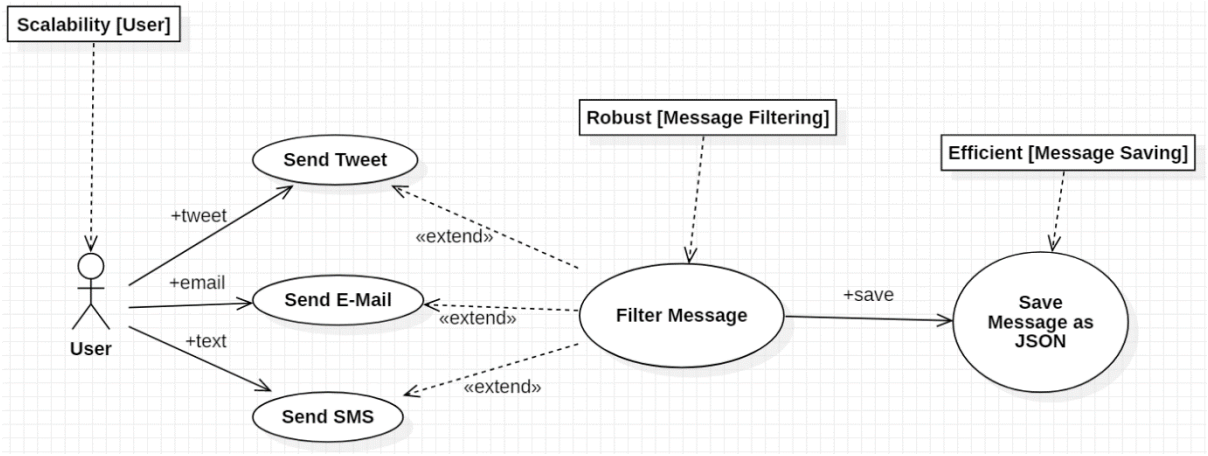

Fig. 3. A diagram of a Local VCS [6]

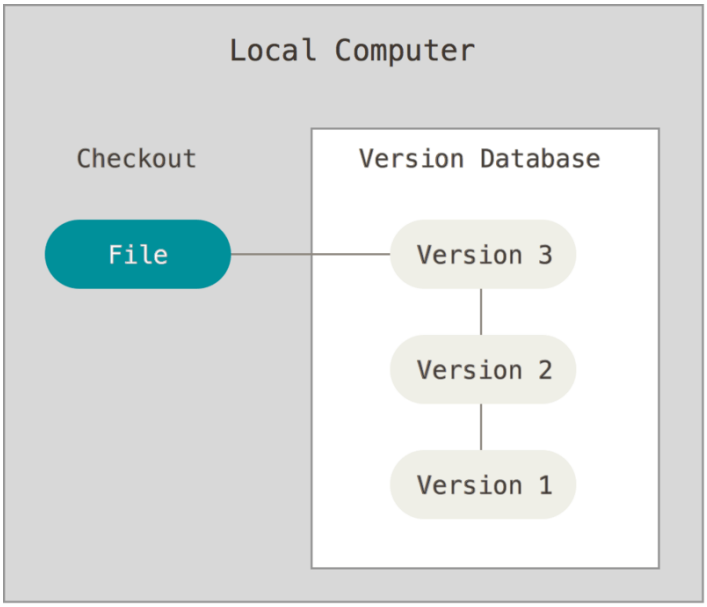

Fig. 4. A diagram of a Centralised VCS [6]

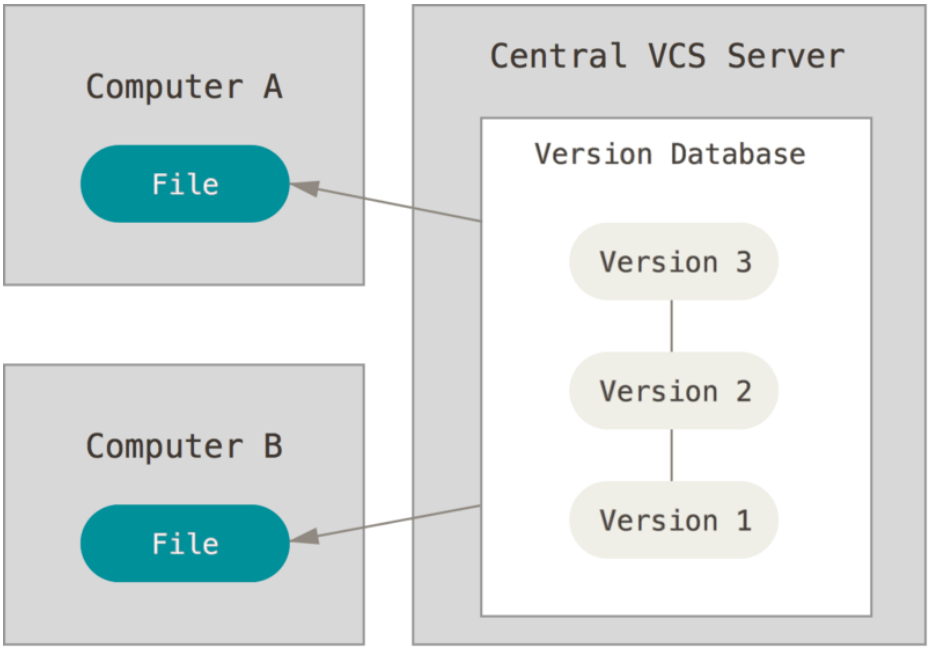

Fig. 5. A diagram of a Distributed VCS [6]

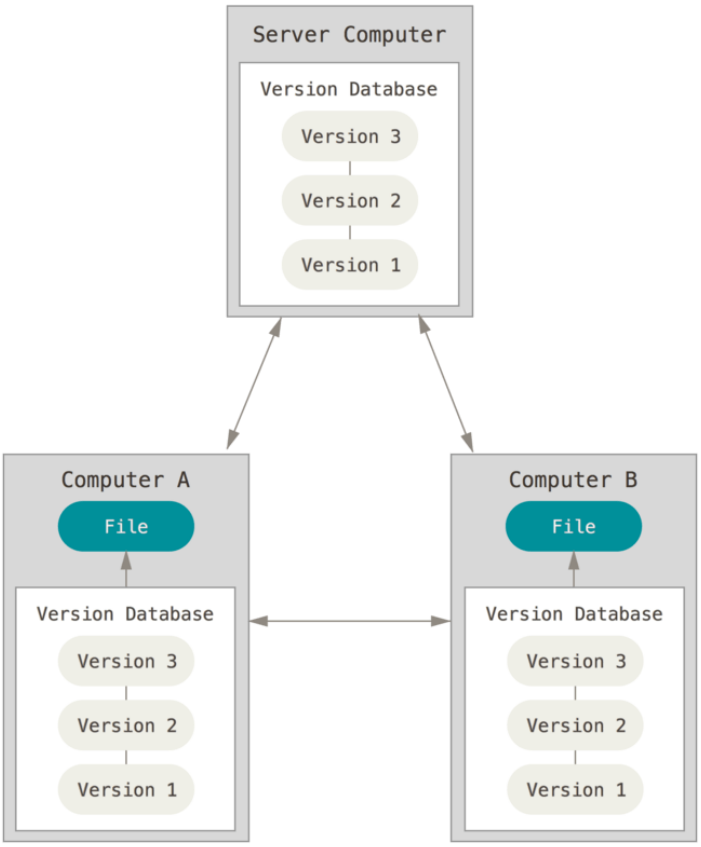

Fig. 6. A summary of all unit tests passing.

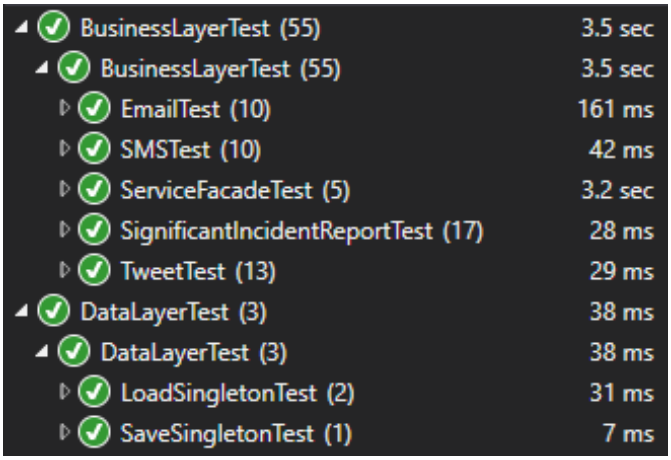

Table 1. The expected results from the Unit Tests

Input	Expected Result	Pass	Resolution/Notes (if applicable)
Email#Validate()	Pass	YES	Ensures the header, subject and text are valid pieces of data.
Email.Header.Length == 10	Pass	YES	
Email.Header.StartsWith(“E”)	Pass	YES	
Regex.IsMatch(Email.Sender, pattern)	Pass	NO	Added the correct pattern to regex
Regex.IsMatch(Email.Sender, pattern)	Pass	YES	(see above)
Email.Subject.Length <= 20	Pass	YES	
Email.Text.Length <= 1024	Pass	YES	
Email("_e.Header", _e.Sender, _e.Subject, _e.Text)	Fail	YES	Throws appropriate exception – Invalid Header
Email(_e.Header, "_e.Sender", _e.Subject, _e.Text)	Fail	YES	Throws appropriate exception – Invalid Sender
Email(_e.Header, _e.Sender, "this is longer than 20 characters", _e.Text)	Fail	YES	Throws appropriate exception – Invalid Subject
Email(_e.Header, _e.Sender, _e.Subject, msg)	Fail	YES	Throws appropriate exception (msg is over 1024 chars)
ServiceFacade#ProcessEmail()	Pass	YES	Object produced is not null
ServiceFacade#ProcessSIR()	Pass	YES	Object produced is not null
ServiceFacade#ProcessSMS()	Pass	YES	Object produced is not null
ServiceFacade#ProcessTweet()	Pass	YES	Object produced is not null
ServiceFacade#Save()	Pass	YES	File created is not null / has contents.
SignificantIncidentReport#Validate()	Pass	YES	
SignificantIncidentReport.Header.Length == 10	Pass	YES	
SignificantIncidentReport.Header.StartsWith(“E”)	Pass	YES	
Regex.Match(SignificantIncidentReport.Sender,pattern)	Pass	NO	Added correct regex pattern
Regex.Match(SignificantIncidentReport.Sender, pattern)	Pass	YES	(see above)
SignificantIncidentReport.Length <= 20	Pass	YES	
SignificantIncidentReport.StartsWith(“SIR”)	Pass	YES	
SignificantIncidentReport.Text.Length <= 1024	Pass	YES	
SignificantIncidentReport("_sir.Sender", _sir.Subject, _sir.Header, _sir.Text)	Fail	YES	Throws appropriate exception – Sender invalid

SignificantIncidentReport(_sir.Sender, "AHHHHHHHH", _sir.Header, _sir.Text)	Fail	YES	Throws appropriate exception – Invalid sender
SignificantIncidentReport(_sir.Sender, _sir.Subject, "_sir.Header", _sir.Text)	Fail	YES	Throws appropriate exception – Invalid header
SignificantIncidentReport(_sir.Sender, _sir.Subject, _sir.Header, msg)	Fail	YES	Throws appropriate exception – msg is longer than 1024 chars
_sir.Code != null	Pass	YES	
_sir.Nature != null	Pass	YES	
_sir.Code.Length == 8	Pass	YES	
_sir.Nature.Length > 3	Pass	YES	
SignificantIncidentReport("40345422@live.napier.ac.uk", "SIR - 26/11/2008", "E326467812", "Sort Code: 99-1-34 Nature of Incident: Theft blah blah blah https://www.google.com/jeff_bezos")	Fail	YES	Throws appropriate exception – The sort code is invalid
SignificantIncidentReport("40345422@live.napier.ac.uk", "SIR - 26/11/2008", "E321467852", "Sort Code: 99-13-34 Nature of Incident: blah blah blah https://www.google.com/jeff_bezos")	Fail	YES	Throws appropriate exception – The nature of the incident is invalid.
SMS.Sender.StartsWith("+")	Pass	YES	
SMS.Sender.Length > 6	Pass	YES	Shortest phone numbers in the world are 6 characters long.
SMS.Header.StartsWith("S")	Pass	YES	
SMS.Header.Length == 10	Pass	YES	
SMS.Header.Substring(1), out _	Pass	YES	This was to ensure the 9 digits after the S were numbers
SMS.Text.Length <= 140	Pass	YES	
SMS(sms.Sender, sms.Header, msg)	Fail	YES	Throws appropriate exception – msg parameter is longer than 140 chars
SMS(sms.Sender, "sms.Header", sms.Text)	Fail	YES	Throws appropriate exception – Header invalid
SMS("sms.Sender", sms.Header, sms.Text)	Fail	YES	Throws appropriate exception – Sender invalid
convert == (word + " <" + _ls.GetAbbreviations()[word] + ">")	Pass	YES	The convert variable was a string containing abbreviations, the test ensured the value was expanded with the parenthesis appended appropriately to the message.
Tweet.Sender.StartsWith("@")	Pass	YES	
Tweet.Header.StartsWith("T")	Pass	YES	
Tweet.Header.Length == 10	Pass	YES	

Tweet.Header.Substring(1), out _	Pass	YES	This was to ensure the 9 digits after the T were numbers
Tweet.Text.Length <= 140	Pass	YES	This software developed is outdated, tweets are now 280 characters long.
Tweet#ExtractHashtags() != null	Pass	YES	
Tweet#ExtractHashtags().Count > 0	Pass	YES	
Tweet != null	Pass	YES	Object itself should not be null
Tweet#ExtractMentions() != null	Pass	YES	
Tweet#ExtractMentions().Count > 0	Pass	YES	
Tweet("refracc", _tweet.Header, _tweet.Text)	Fail	YES	Throws appropriate exception – Sender is invalid
Tweet(_tweet.Sender, "_tweet.Header", _tweet.Text)	Fail	YES	Throws appropriate exception – Header invalid
Tweet(_tweet.Sender, _tweet.Header, msg)	Fail	YES	Throws appropriate exception – msg variable above 140 characters.
_ls == _ls1	Pass	YES	_ls and _ls1 are both instances of the LoadSingleton class
LoadSinglton.abbreviations != null	Pass	YES	
LoadSingleton.abbreviations.Count > 0	Pass	YES	
_ss == _ss1	Pass	YES	_ss and _ss1 are both instances of the SaveSingleton class