

Submitted for the Degree of B.Sc. in Computer Science, 2018-2019

Distributed and Decentralised Issue Tracking with Git

Registration Number: 201645569

Name: Jack Maclauchlan

"Except where explicitly stated all the work in this report, including appendices, is my own and was carried out during my final year. It has not been submitted for assessment in any other context. "

Signature _____ Date _____

"I agree to this material being made available in whole or in part to benefit the education of future students."

Signature _____ Date _____

Abstract

Git is a distributed version control system used primarily for source-code management. One of the main features of Git is that it is decentralised. Each repository is self-contained, and data can be synchronised between repositories by pushing and pulling, without the need for a central repository.

GitHub is a popular Git repository hosting service that developers commonly use for the convenience of sharing changes to the project codebase over the internet. Services like GitHub often offer their own features such as issue tracking and continuous integration pipelines. The problem with using an issue tracking system on GitHub or a similar service is that it breaks the decentralised nature of Git.

This project aims to create an ITS that takes advantage of Git's distributed and decentralised nature. Issues will exist within the self-contained Git repository giving every developer a full copy of the issue history. These issues will then be able to be pushed, pulled, merged, and synchronised along with all the other data stored in the repository.

The system was successfully implemented. The objectives identified for the project were accomplished and the resulting system was evaluated to be fit for purpose.

Acknowledgements

I would like to express my gratitude to my supervisor, Robert Atkey, for the continuous support and guidance during the project.

I would also like to thank my friends who took the time to proof read my report.

Contents

Chapter 1 – Introduction.....	1
1.1 Problem Statement.....	1
1.2 Project Objectives	1
1.3 Project Outcome	2
1.4 Project Structure	2
Chapter 2 – Background Research and Related Work.....	3
2.2 Git.....	3
2.3 Git Workflows	4
2.3.1 Centralised Workflow	4
2.3.2 Decentralised Workflow	5
2.4 Issue Tracking Systems.....	6
2.4.1 Jira	6
2.4.2 GitHub Issues	7
2.5 Related work	8
2.5.1 Git-appraise.....	8
2.5.2 Git-issue	8
2.5.3 Git-bug	9
2.5.4 Git-dit	10
Chapter 3 – Problem Description and Specification	11
3.1 Problem Description	11
3.2 Requirements Gathering.....	11
3.3 Specification.....	13
3.3.1 Original Functional Requirements	13
3.3.2 Final Functional Requirements	14
3.3.3 Non-Functional Requirements.....	14
3.3.4 Use cases.....	15
3.4 Project Plan	15
3.4.1 Design Methodology.....	15
3.4.2 Planned Development Sprints	16
3.5 Marking Scheme	16
Chapter 4 – System Design	17
4.1 User Interface	17
4.2 System Architecture.....	18
4.3 Data Management	19

4.4	Distributed Issue Tracking System	22
Chapter 5 – Detailed Design and Implementation		23
5.1	Implementation Language	23
5.1.1	Java.....	23
5.1.2	Python	23
5.1.3	Go.....	24
5.1.4	Chosen Language	24
5.2	Libraries Used.....	24
5.2.1	JGit	24
5.2.2	Commons CLI	25
5.2.3	Jackson	25
5.2.4	OpenCSV	25
5.2.5	Commons IO.....	25
5.3	Development Tools	25
5.3.1	Maven	25
5.3.2	IntelliJ IDEA Ultimate	25
5.4	Strategy Design Pattern	26
5.5	Challenging/Interesting Aspects of Implementation	27
5.5.1	Storing Issues	27
5.5.2	Pulling Issues	31
5.5.3	Query Language	31
5.6	Problems Encountered	33
5.6.1	Dirty Work Tree.....	33
5.6.2	Formatting the CSV files.....	34
5.6.3	Nicknames.....	35
Chapter 6 – Verification and Validation.....		36
6.1	Verification.....	36
6.2	Validation	38
6.2.1	Black Box Testing.....	38
6.2.2	White Box Testing	39
6.2.3	Testing Technologies.....	39
6.2.3.1	Mockito	39
6.2.3.2	Junit Pioneer	39
6.2.3.3	Pi-test	40
Chapter 7 – Results and Evaluation		41
7.1	Evaluation Criteria.....	41

7.2	System Comparison	41
7.3	System Performance	42
7.3.1	Benchmark Testing.....	42
7.3.2	Benchmarking Conclusion.....	43
7.4	Potential Usage of System	44
7.4.1	Workflows	44
7.4.1.1	Centralised Workflow	44
7.4.1.2	Decentralised Workflow	44
7.4.2	Other Systems	45
7.5	Portability Evaluation.....	45
7.6	Evaluation Conclusion	46
Chapter 8 – Summary and Conclusion		47
8.1	Project Summary.....	47
8.2	Future Work	47
8.2.1	Git Hooks.....	47
8.2.2	Signed Commits	47
8.2.3	Survey with Expert Users	48
8.2.4	Invert Option in Query Language.....	48
8.2.5	Git Annex.....	48
8.3	Conclusion.....	48
Appendix A – References		1
Appendix B – Detailed Specification and Design		4
B.1	Sprint details	4
B.2	Project Plan	7
B.2.1	Original Project Plan.....	7
B.2.2	Updated Project Plan	8
B.3	Use Cases	9
Appendix C – Detailed Test Strategy and Test Cases		20
C.1	Black Box Tests	20
C.2	White Box Tests	20
Appendix D—System Class Diagram		22
Appendix E – User Guide and Quick Start Guide		23
E.1	Quick Start Guide	23
E.2	User Guide	24
Appendix F – Installation Guide		28
F.1	Obtaining a JAR from Source Code	28

F.2	Setting up JAR to use.....	28
	Appendix G – Maintenance Guide	29
G.1	Adding a Feature.....	29
G.2	Creating a Command	29
	Appendix H – Command Help Texts	30

Chapter 1 – Introduction

1.1 Problem Statement

Git is a distributed version control system (DVCS) used primarily for source-code management. Every Git directory contains a full copy of a project's development history. These git directories are their own fully-fledged and self-contained repositories that can be shared from one repository to another by pushing and pulling changes, eliminating the need for a central repository.

Software development teams often use a hosted Git repository service like GitHub/GitLab for the convenience of sharing changes to the codebase over the internet. These services add their own functionality on top of Git's DVCS such as issue tracking, pull-requests and even continuous integration and continuous delivery pipelines.

The problem with using an issue tracking system (ITS) on GitHub or a similar service is that it breaks the decentralised nature of Git. These issues are stored only on the hosted repository making it a centralised system and to use the feature you must do so via GitHub.

1.2 Project Objectives

The objective of the project is to create an ITS that takes advantage of Git's distributed and decentralised nature. The ITS will use a command line interface and provide the following functionality: create, edit, and delete issues; open and close issues; add tags/media/watchers/assignees to issues; display issue information; and comment on issues. Issues will exist within the self-contained Git repository meaning every developer with a local copy of the repository will have a full history of issues. These issues will then be able to be pushed, pulled, merged and synchronised along with all the other data stored in the repository.

The system will be user-friendly, reliable, and robust. The CLI will be easy to use with well documented manuals containing example usage. During synchronisation of issues, any data contained in the repository must not be lost or incorrectly modified. The system must be fit for purpose when compared with other issue tracking systems.

1.3 Project Outcome

Overall, the project was a success. All objectives were completed with several additional features were added in order to improve the possible usage of the system. The system was evaluated to be fit for purpose.

1.4 Project Structure

Following this chapter, the report structure is as follows:

Chapter 2 – Related Work: Researches similar systems and concepts related to the project.

Chapter 3 – Problem Description and Specification: Defines the projects requirements and project plan.

Chapter 4 – System Design: Provides a high-level overview of the system design.

Chapter 5 – Detailed Design and Implementation: Describes the system design and implementation in depth, explaining interesting parts of the code and design patterns used.

Chapter 6 – Verification and Validation: Discusses how the project requirements are fulfilled and an overview of the system testing.

Chapter 7 – Results and Evaluation: Evaluates if the system is fit for purpose.

Chapter 8 – Summary and Conclusion: Provides a summary of the system, future work that could be carried out and a conclusion for the project.

For all figures contained in the report, a full-size image can be viewed in the submission files (figures/figure*).

Chapter 2 – Background Research and Related Work

This chapter aims to provide an overview and understanding of the concepts important to implementing a distributed and decentralised system. Popular issue tracking systems such as Jira and GitHub are researched in order to highlight the main features and functionalities of this type of system. Several distributed and decentralised systems were researched to investigate how they handle the constraints of having no centralised server.

2.1 Distributed and Decentralised Systems

A distributed system consists of several components that are stored on different computers but run together as a single system (IBM, 2019). The computers can be connected in a variety of ways such as local area networks or over the internet. The system can consist of any number of computers and will interact with each other in order to achieve a common goal. The way the components communicate and make decisions is dependent on if the system is centralised or decentralised.

A centralised system is one that contains a component that all others are connected to. This central component will have direct control over the full system and makes all decisions. The processing may be done on any of the connected components.

A decentralised system is when there is no single component that makes all the decisions. All components make decisions for their own actions to accomplish the systems goal.

2.2 Git

Git is a Distributed Version Control System (DVCS) used primarily for managing source code in software development projects. Git is distributed as every user with a copy of the repository will have an entire copy of the project history. The benefits of this are that every user's repository serves as a backup and that there is no single point of failure (Git, 2019).

There are other version control systems such as Apache Subversion (SVN) (Apache, 2019) which are not distributed. The main difference between Git and SVN is that every developer using Git has a full local copy of the repository and history meaning they can work offline. Developers using SVN connect to a server and only have a local copy of the files they are working on. The developer must always be connected to the server. SVN repositories also have a linear history, every commit comes one after the other. While it is possible to have a linear history in Git, branches with independent commit histories are very commonly merged into the master branch making the history non-linear.

When users want to share their changes to a project, they do so in two ways: push and pull. These are explicit ways for a user to send or receive updates to the project. This is unlike a service such as Dropbox (Dropbox, 2019) which will constantly synchronise data. The push command will send changes to the local repository to a remote repository. The pull command checks for any changes to the project on the remote repository and will update the local repository with them if necessary.

2.3 Git Workflows

Git supports both centralised and decentralised workflows. A workflow is a recommendation for how developers should use Git to collaborate with each other, i.e. share their changes to the project.

2.3.1 Centralised Workflow

A centralised Git workflow is where users push changes to a shared repository. This repository is usually hosted by a service like GitHub (GitHub, 2019) or GitLab (GitLab, 2019). Other users can then pull these changes to their own local repository. An example of a centralised workflow can be seen in the figure below:

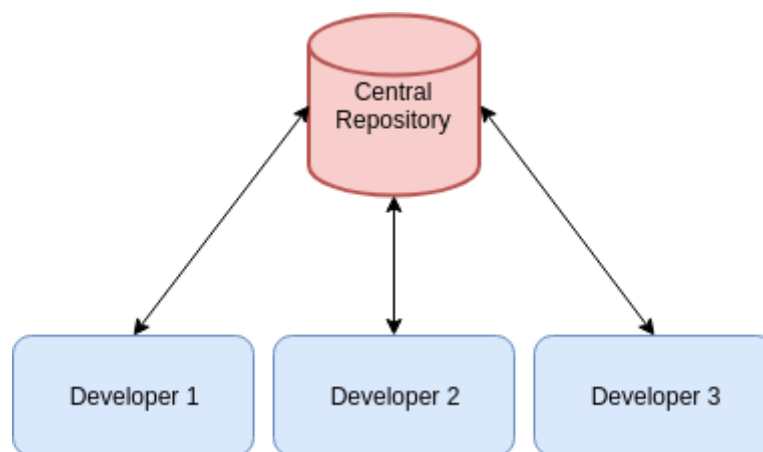


Figure 1 - Centralised Workflow

If Developer 2 has pushed changes to the central repository since the last time Developer 1 fetched, Git will reject Developer 1's changes until they synchronise with the Developer 2's changes (Git, 2019). This type of workflow is common in small development teams where developers are comfortable with everyone directly making changes to the central repository.

2.3.2 Decentralised Workflow

A decentralised Git workflow is where there is no central repository/single source of truth. Users can synchronise their repositories with each other when it is convenient for them. There are many types of decentralised workflows which are made possible by Git being distributed.

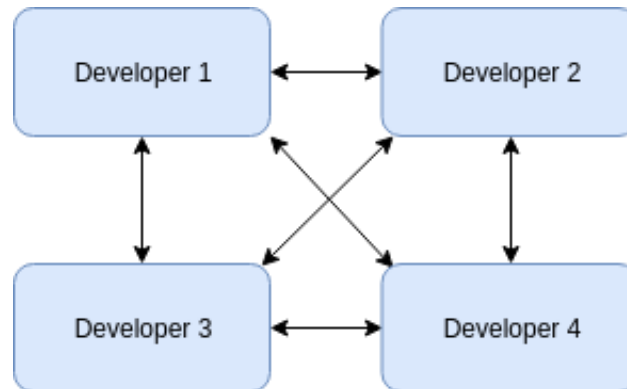


Figure 2 - Decentralised Workflow

Figure 2 shows a decentralised workflow where a developer can share their changes with any other developer. Git makes this possible because each repository can have multiple remotes with developers having read access to everyone else's repository (Git, 2019).

Another kind of decentralised workflow that is popular is using an Integration Manager:

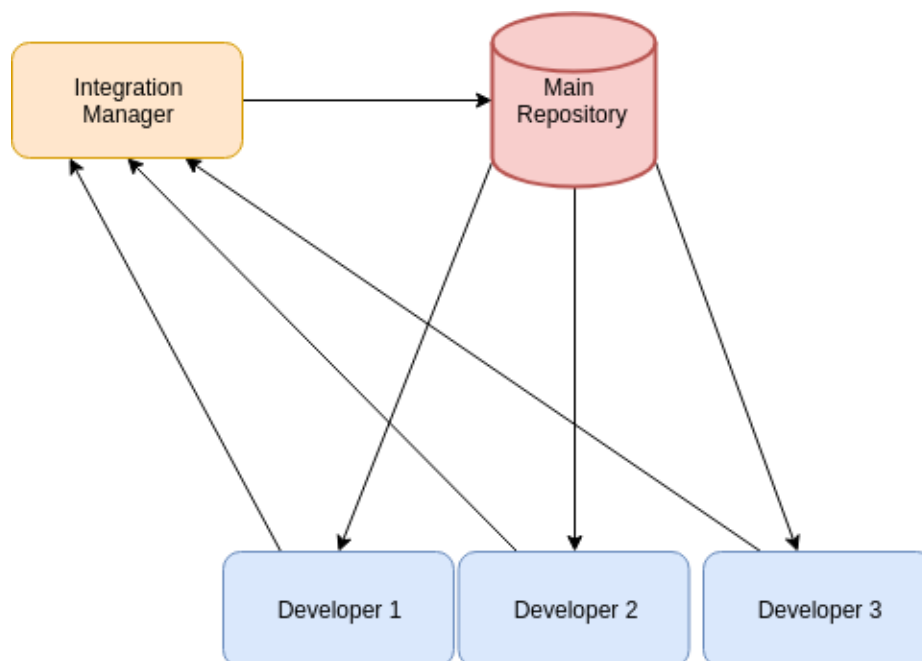


Figure 3- Integration Manager Workflow

Developers can receive updates from the main repository but any changes they make must go through the Integration Manager. This helps with issues such as code quality. The integration manager can review all changes before they are pushed to the main code repository.

2.4 Issue Tracking Systems

An issue tracking system (ITS) is a software application used by teams to report and track issues related to their work. ITSs can be used in many different types of teams or organisations such as call centres and IT help desks to manage customer and employee requests. ITSs are very common in software development teams. They provide a place for developers to report bugs/problems, suggest improvements/new features and organise tasks related to a project. The common attributes of an issue are a title, description, status (e.g. open or closed), an assignee (the person to resolve the issue), watchers (users who receive notifications when the issue is updated), comments, and a tag or label (used to categorise the issue, e.g. “bug”, “feature” or “enhancement”). Popular issue tracking systems used in software development teams are Jira by Atlassian and Issues by GitHub. The issues created in these systems contain all the basic attributes listed here but also provide some additional features to aid with project management or to integrate with other services they provide.

2.4.1 Jira

Jira (Atlassian, 2019) was originally created as an issue tracking system that was specifically targeted towards software development teams. It has now evolved into much more than just an issue tracking system, mainly a project management tool used by agile teams. A typical Jira issue, as shown in figure 4, contains the attributes listed in the previous section and several more. This issue is a feature request for an existing application to alert the user when the weather changes. The attributes of this issue not listed above are that it contains a place for attachments (e.g. images), a place to link a development branch, a customisable status (e.g. “In progress” – more descriptive than open/closed), a story point estimate and a time tracking feature. (High quality image in submission files – figures/figure4).

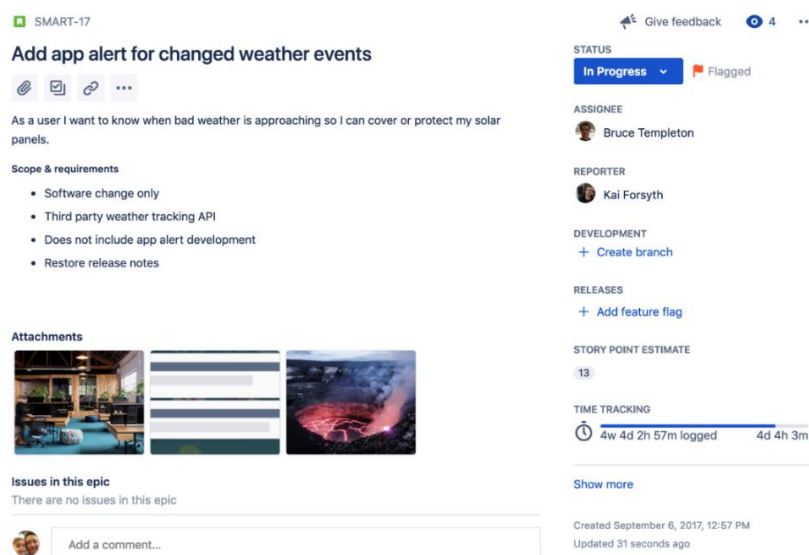


Figure 4 - Jira Issue (Atlassian, 2019)

2.4.2 GitHub Issues

Ivy Renderer (beta) #21706

Open IgorMinar opened this issue on 22 Jan 2018 · 113 comments

IgorMinar commented on 22 Jan 2018 • edited by mhevery

Member + 👤 ...

Overview

Ivy is a new backwards-compatible Angular renderer focused on further speed improvements, size reduction, and increased flexibility.

Ivy is currently not feature complete, but can be tested via `enableIvy: true` `angularCompilerOptions` flag.

We currently expect Ivy to remain behind the flag until it's feature complete and battle tested at Google. In the meantime you can check out this [Hello World demo](#).

To see status of Ivy implementation go [here](#).

Related Issues:

- [Tree-Shakeable Providers Followup Work](#)

👍 487
😄 87
🔥 166
😬 6
❤️ 208
👁️ 2

↑

👤 IgorMinar added this to the v6.0 milestone on 22 Jan 2018

👤

👤 IgorMinar assigned mhevery on 22 Jan 2018

✕

🚫 angular deleted a comment from IgorMinar on 23 Jan 2018

🏷️

👤 mhevery added comp: ivy type: feature labels on 24 Jan 2018

✕

🚫 angular deleted a comment from LinBoLen on 26 Jan 2018

✕

🚫 angular deleted a comment from arjunyel on 26 Jan 2018

Assignees

👤 mhevery

Labels

comp: ivy

type: feature

Projects

None yet

Milestone

v7.0

Notifications

🔔 Subscribe

You're not receiving notifications from this thread.

57 participants

and others

Compared to Jira, GitHub's Issues do not contain as many project management features; however, there is the option to create project boards which are similar to Jira's scrum boards and also generate basic statistics for issues using the "Pulse" feature. GitHub provides a powerful search tool to filter through issues from specific repositories or the whole of GitHub. There is also the ability to attach pull requests to an issue so that it can be automatically closed when the pull request is merged.

2.5 Related work

The reason the following systems were chosen for research is that they are all distributed systems that have no centralised server. As there is no authoritative server it creates constraints for this type of system. In order to synchronise with other repositories the data model each system uses must be able to identify unique objects and handle the merges correctly. Each system researched has a different method of storing and synchronising data, so it will provide valuable insight for implementing this project's issue tracking system.

2.5.1 Git-appraise

Git-appraise (Google, 2019) is a distributed code review system for Git repositories. Code reviews are used to decide if code on a branch is ready to be merged into another. The system stores a copy of the code review history inside the Git repository meaning every developer with access has a copy; therefore, eliminating the need for a central server. When synchronising the repositories with others, updates are merged automatically by the tool.

The data for the code reviews are stored in git-notes (Git, 2019). Git-notes are blobs (see 5.5.1) that are attached to objects in a repository, without touching the object themselves. The objects that the notes are attached to by default and in Git-appraise are commit messages. Each note consists of a single line of JSON. This allows for the automatic Git-note merge strategy "cat_sort_uniq" to be used. This merge strategy removes duplicate lines.

The CLI for this system is also very intuitive so will provide a standard to hold the interface of the issue tracking system to. All user input required by the tool is given as command line arguments and no text editor is opened.

2.5.2 Git-issue

Git-issue (Spinellis, 2019) is a decentralised issue tracking system using Git. This project takes a minimalist approach and uses only one shell script for the entire application. The system contains all the standard features of an issue tracking system: issue title, description, comments, attachments, tags, watchers, and assignees.

The data model used by the system for storing issues is text files. Text files are simple to work with and can be viewed, edited, shared or backed-up easily with other programs if the user wants. The issues are stored in a hidden Git repository called ".issues" and inside will contain a new directory for each issue. These issue directories will then contain text files containing data such as description, comments, etc. The names of each issue directory within ".issues" is the SHA of the issue's initial commit. The issues can then be shared normally through Git which will leave all the issues within Git's version history.

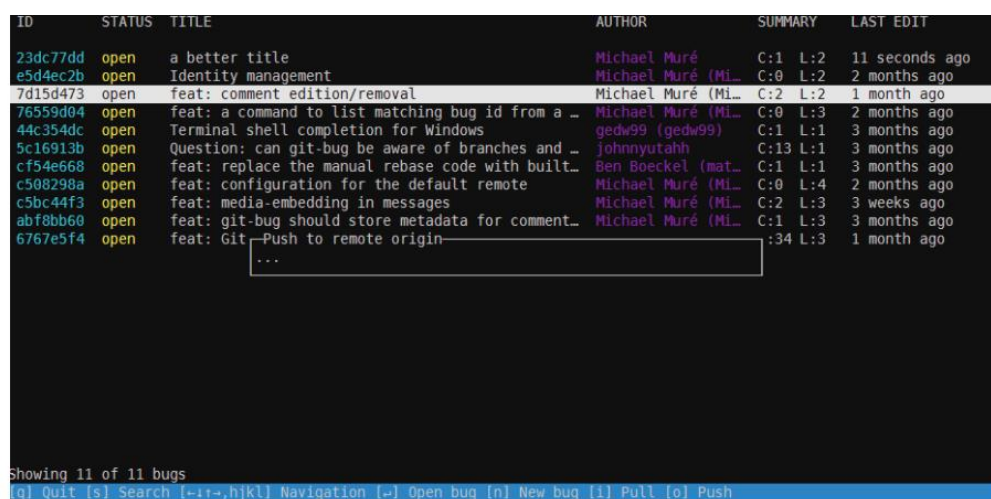
Git-issue's CLI is also highly intuitive using simple commands that are like regular Git commands. Details of an issue can be input directly to the terminal using arguments or a user can choose to open the issue details in an editor of their choice.

2.5.3 Git-bug

Git-bug (Muré, 2019) is a distributed bug tracker which is built on top of Git. The project is almost identical to a standard issue tracking system, only lacking a few features. A bug (essentially an issue) can be created with a title, description, tags, attachments, and comments. The features it lacks are the ability to give a bug watchers and assignees.

For storing data about bugs, the system creates a new branch "bugs" inside the repository. On this branch there is a new directory for each bug. The ID of the bug is used as the directory name. The unique ID of the bug is the hash value of the bug's information (title, description, timestamp). This ID will stay with the bug even if it is edited. Inside of these directories is a chain of commits containing a tree. Each tree contains blobs which are serialised golang structs of the bug details and sub-trees which will hold things such as media blobs that are included in the bug. Each commit in the chain is an edit a user has made to the bug e.g. adding an updated description.

Git-bug includes three user interface options: an interactive terminal UI, a web UI and a normal CLI. The interactive terminal provides an extremely simple way to use the system. There are multiple screens providing convenient ways to create, edit, view, and share bugs. The user navigates around by using arrow keys and information about each bug is displayed clearly in rows and columns.



ID	STATUS	TITLE	AUTHOR	SUMMARY	LAST EDIT
23dc77dd	open	a better title	Michael Muré	C:1 L:2	11 seconds ago
e5d4ec2b	open	Identity management	Michael Muré (Mi...	C:0 L:2	2 months ago
7d15d473	open	feat: comment edition/removal	Michael Muré (Mi...	C:2 L:2	1 month ago
76559d04	open	feat: a command to list matching bug id from a _	Michael Muré (Mi...	C:0 L:3	2 months ago
44c354dc	open	Terminal shell completion for Windows	gedw99 (gedw99)	C:1 L:1	3 months ago
5c16913b	open	Question: can git-bug be aware of branches and _	johnnyutahh	C:13 L:1	3 months ago
cf54e668	open	feat: replace the manual rebase code with built_	Ben Boeckel (mat_	C:1 L:1	3 months ago
c508298a	open	feat: configuration for the default remote	Michael Muré (Mi...	C:0 L:4	2 months ago
c5bc44f3	open	feat: media-embedding in messages	Michael Muré (Mi...	C:2 L:3	3 weeks ago
abf8bb60	open	feat: git-bug should store metadata for comment_	Michael Muré (Mi...	C:1 L:3	3 months ago
6767e5f4	open	feat: Git-Push to remote origin		:34 L:3	1 month ago

Showing 11 of 11 bugs

[q] Quit [s] Search [-tr-,h,jkl] Navigation [w] Open bug [n] New bug [i] Pull [o] Push

Figure 6- Interactive Terminal (Muré, 2019)

The web UI serves static content through a local HTTP server. The layout is simplistic but clearly shows information about the bugs. It is currently still in early phases on development so the developer will most likely improve the quality.

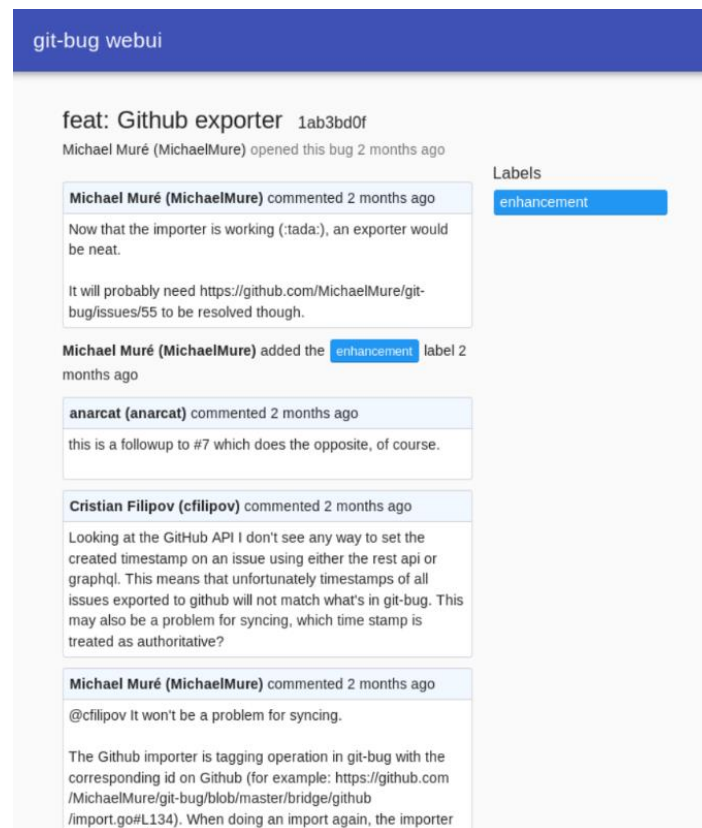


Figure 7- Git-bug Web UI (Muré, 2019)

The final option is to just use basic command line arguments which will open a text editor for the user to enter details about the bug.

2.5.4 Git-dit

Git-dit (Ganz & Beyer, 2019) is another decentralised issue tracking system. The functionality of this system is limited to that of other issue tracking systems. The features an issue can have are a title, description, comment, and a tag.

No structured data is used in this system for storing issues. Simply an issue branch is created and then for every new issue a directory is made containing a commit with the issue details in the commit message. Comments are stored inside sub-directories also using commit messages. There is a strict format for the commit messages so that the system can identify what is the title, description, and tag.

The CLI in this project also uses commands similar to regular git commands meaning that, (although it lacks features compared to the others) it is still simple to use. This project is not as complete as the three projects above but is still useful for referencing the data model.

Chapter 3 – Problem Description and Specification

3.1 Problem Description

Git is a distributed version control system (DVCS) used primarily for source-code management. Every Git directory contains a full copy of a project's development history. These git directories are their own fully-fledged and self-contained repositories that can be shared from one repository to another by pushing and pulling changes, eliminating the need for a central repository.

Software development teams often use a hosted Git repository service like GitHub/GitLab for the convenience of sharing changes to the codebase over the internet. These services add their own functionality on top of Git's DVCS such as issue tracking, pull-requests and even continuous integration and continuous delivery pipelines.

The problem with using an issue tracking system (ITS) on GitHub or a similar service is that it breaks the decentralised nature of Git. These issues are stored only on the hosted repository making it a centralised system and to use the feature you must do so via GitHub.

This project aims to create an ITS that takes advantage of Git's distributed and decentralised nature. Issues will exist within the self-contained Git repository meaning every developer with a local copy of the repository will have a full history of issues. These issues will then be able to be pushed, pulled, merged and synchronised along with all the other data stored in the repository. The system will also contain all the features and attributes of a standard ITS mentioned in the previous section of this report.

3.2 Requirements Gathering

When gathering requirements for this project it was important to first capture the minimum requirements of an issue tracking system and then expand on them within reason. This is because the scope of this type of system can be expanded infinitely. Requirements were gathered from three places.

The first-place requirements were created from was the original project description released by my supervisor Dr Robert Atkey. The requirements relating to how the system should be distributed and decentralised, and some basic features of an ITS like being able to create and update issues were given here.

The next place requirements were gathered from was the research of related systems (section 2.4). ITSs such as Jira and Issues (GitHub) were studied and their similarities were highlighted. This made it clear which attributes an issue should have and the common features that made use of issues.

The final place requirements were gathered from was a survey of issue tracking systems done by JetBrains in 2016 to promote their own ITS “YouTrack” (JetBrains, 2016). There is a disclaimer by JetBrains saying that the survey was primarily completed by customers or people related to JetBrains; therefore, it is possible that there may be some bias in the survey results. The main takeaways from the survey were that Jira ranked highest in popularity and in a comparison of features. The highest rated features of Jira were that it was easy to work with issues, the reporting tools were effective and that it was easy to track time. GitHub Issues ranked not far behind Jira with its best features being the simplicity of working with issues, the searching capabilities, and the ability to integrate it with other systems.

3.3 Specification

3.3.1 Original Functional Requirements

- Users can:
 - Initialise an issue repository inside a pre-existing Git repository
 - Create an issue with a title, description, and a default open status
 - Close issues
 - Reopen closed issues
 - Add multiple tags to issues
 - Remove tags from issues
 - Add assignees to issues (assignees are users who must solve the issue)
 - Remove assignees from issues
 - Add watchers to issues (watchers are users who will receive notifications about updates to the issue)
 - Remove watchers from issues
 - Add media attachments to issues (e.g. pdfs and images)
 - Remove media attachments
 - Edit the title of an issue
 - Edit the description of an issue
 - Comment on issues
 - View the full details of an issue
 - View comments on issues
 - List all issues – displaying the id and title of all issues
 - List all issues with a verbose option displaying all the information about the issue
 - Synchronise their repositories with others using push and pull commands
- Issues are stored on their own branch inside a Git Repository
- Issues track extra attributes like creation time, last edit time and who created the issue originally
- In the event of a merge conflict when synchronising repositories, users are presented with a way to fix the merge or abandon it
- System is distributed like Git – does a clone of the entire repository (full copy of issue history included)
- System is decentralised – No need for a central server to synchronise, independent repositories can clone and synchronise with each other

3.3.2 Final Functional Requirements

Midway through the project additional requirements were added to the project. The reason these features were added is because they provide many additional ways that the system can be used. Mainly so that it is easier to use or can be used by other programs such as a data analyser or report creator. The final list of functional requirements includes all listed above and the following:

- Users can filter which issues they see by using a query language
- Query language can filter issues by title, description, status, tags, watchers, assignees, creators, edit time and creation time
- Create CSV files containing information on issues
- Create CSV files containing information on comments
- Data contained in CSV files can be filtered using the query language
- Users can add a unique nickname to issues (makes the issue easier to identify in terminal)
- Users can update an issue's nickname
- Users can remove an issue's nickname

Several more requirements were thought of and attempted; however, due to lack of support with Java libraries, they were not included with the final version of the system. The details of these requirements can be found in chapter 8.2.

3.3.3 Non-Functional Requirements

- Uses command line interface (CLI)
- Compatible with as many operating systems as possible, minimum requirement is to run on Linux based operating systems using a bash shell
- CLI is intuitive –commands are simple and rejects invalid inputs with useful error messages
- Well documented help menu and manual with example usage
- Query language is simple to use
- Issue data is displayed neatly in the terminal window
- System is reliable and robust – data must not be lost or incorrectly modified (for both issue data and data related to the project contained in the repository)
- Data is stored efficiently as there will potentially be many issues inside each repository

3.3.4 Use cases

Using the requirements of the project a list of use cases was created. The reason they were created was to help better understand the requirements and the interaction between the user and the system. The normal expected behaviour was defined and how to handle any alternate behaviour. The format of the use cases is based on Martin Fowlers template (Fowler, 2003). Each use case defines actors involved in the process; the pre and post conditions; a description or story for the use case; the expected normal flow; and any alternate flows. The full list of use cases can be found in appendix B.3.

3.4 Project Plan

3.4.1 Design Methodology

The original design methodology planned for the project was to use an iterative waterfall model however, it was decided that this would be an inappropriate choice for this type of system and an agile methodology was used instead. Before this project I had no experience in developing distributed or decentralised systems. Therefore, I would have been unable to define an accurate waterfall project plan as I did not know what implementing these requirements would entail. Several constraints are introduced when developing this type of system as there is no central authoritative server to make decisions. Consequently, the design methodology had to be flexible enough to allow a reasonable amount of trial and error in finding the best approach to store and synchronise data.

A waterfall model would have been effective had the requirements been fully understood at the beginning of the project but for the reasons mentioned above they are not. Taking an agile approach, several short development sprints were planned to develop a small number of features at a time. At the end of each sprint the results were documented and reflected upon. This ensured no non-optimal decisions were hidden until the end of the project, features could be improved at any time. Using agile also allowed for several new requirements to be added to the project in the middle of development.

Details of the features and requirements that were implemented during each sprint are discussed in the next section. The full project plan including details about the report writing can be found in appendix B.2.

3.4.2 Planned Development Sprints

The following 5 development sprints were planned to develop the system. However, due to the nature of the project small changes were made. These changes are documented in appendix B.1.

The plan for the first development sprint was to create the CLI that the system will use, the ability to create a new branch within a Git repository that handles storing issue data and a basic template for issues. This sprint was scheduled to last one week.

During the second sprint, the plan was to make issues more detailed containing all the required attributes of an issue such as tags, watchers, assignees, media attachments, etc. The ability for the user to fill in these attributes was also included in the implementation plan (e.g. add/remove a user as a watcher on an issue). The ability to list and filter all issues in the repository was also planned to be created during this week-long sprint. The functionality to create and view comments would also be added.

The third sprint was planned to last 4 days. During this period, the ability to synchronise issues with other repositories was to be implemented. The focus was just to make the push and pull commands work for issues that will not create any merge conflicts.

The final sprint for developing the system was planned to last for one week and 3 days. During this time, the functionality to resolve merge conflicts was to be implemented making sure not to lose or incorrectly modify data in the repository.

The last sprint of development in general was to complete testing of the system. Unit tests (white-box testing), integration tests, mutation tests and user tests (black-box testing) were to be carried out during this week.

The details of implementation, any problems and the changes to the project plan that arose during these sprints is documented in appendix B.1.

3.5 Marking Scheme

The marking scheme for this project is “Software Development Based” as the majority of the project was designing, implementing and testing a console application.

Chapter 4 – System Design

4.1 User Interface

When choosing the type of user interface to implement in the system, it was important to choose an interface that appeals to the targeted end users (developers). Therefore, it was decided the system would use a command line interface instead of a GUI. There are several factors that influenced this decision. Most importantly is that the system makes use of Git functionality. There are clients built to provide a GUI for Git, but most developers will just use the command line. Developers will have an open terminal with its working directory inside the project they are working on. This means it is convenient and will be the fastest way for a developer to interact with the ITS as it operates out of the project's directory. Using a CLI is less intuitive than a GUI and takes longer to learn; however, once a user is familiar with the commands it is usually faster.

When designing the commands for the CLI, the requirements were looked at and related attributes were grouped into one command. For example, to add or remove a watcher to an issue, the command starts with “git ddit watcher” with additional options afterwards specifying whether to add or remove a watcher (“--add” or “--remove”). In total there are 14 unique commands in the system. These commands all have required options that need to be specified, if they are not then a suitable error message will be shown to the user.

Every command of the system takes a “--help” or “-h” option which shows a comprehensive help text on the options that the command can use and some example usages. The full list of commands can be found in appendix H along with their help text. In the user guide (appendix E) a quick “getting started” guide can be found followed by a full tutorial on how to use the system.

When implementing the options each command has, several attempts were made to keep them user friendly (full details in appendix B.1). Using a CLI best practices guide (Deisz, 2016), an effort was made to make sure commands were simple to understand and use. All options that could have a default option had one, every option has a short and long name (e.g. “-h” and “--help”), common option names are used (e.g. “-ls” and “-rm”), and no option is dependent on position, so can be specified in any order.

4.2 System Architecture

The software architectural style of the project is object-orientated; therefore, has a focus on data abstraction and code reuse. As a result of this, code in the system is packaged by feature to help keep modules decoupled. Each module that is used by another provides a public interface to be used. Data is not shared directly. This means internal data representations can be changed without affecting the calling code.

The figure below shows an extremely high-level overview of the packages contained in the system. The in-depth design of the system can be found in the next chapter. An arrow from one module to another indicates that the module where the arrow originates from, knows about the others public interface.

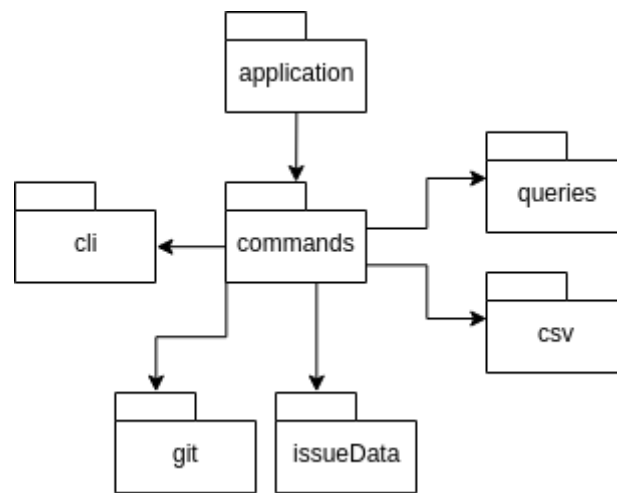


Figure 8 - Package Overview

The “application” package contains only one class which is a driver class for the application. It takes the command line arguments entered by the user and passes them into the system.

The “commands” package is where most classes of the system are contained. When the user’s arguments are passed into the system, they are passed to a class which uses the CLI module to interpret which command of the system the user wants to run. Once the command has been interpreted, it is then instantiated and executed. The class which interprets the command only knows the public interface to the classes inside the CLI module – this is an example of loose coupling; it does not matter to the interpreter class how the classes in the CLI module work internally. The same principle is used with the other classes in the “commands” package when they interact with the “git”, “csv”, and “queries” modules.

Each command of the system has a corresponding class that implements the interface “Command”. These commands are good examples to explain how the SOLID principles are applied everywhere in

the system's design. The solid principles are five design principles intended to improve code quality. An example of the single responsibility principle being applied, is how all classes implementing Command perform only functionality related to that command and nothing else. An example of Liskov substitution principle in the system is, anywhere a command class is invoked it is replaceable with any other class implementing command and will work correctly. The interface segregation principle is applied as every method in the Command interface is implemented within each concrete command class.

The benefits the SOLID design principles bring to the project are that they make the system's design easier to understand, flexible and maintainable. The maintenance guide can be found in appendix G.

4.3 Data Management

As previously mentioned in the report, implementing a distributed system introduces some constraints due to having no central authoritative server. A centralised system provides several benefits such as data integrity. Data is only stored in one place so when it is updated it is available to all users immediately, meaning it is as accurate and consistent as possible. In contrast, in a distributed system, when data is updated in one part of the system it is not instantly available to all other parts. The changes must be synchronised so that all parts of the system are consistent.

This is important for this project because any user can create, edit, and delete issues at any time (including when the user is working offline). When synchronising the different repositories, the system must be able to identify unique issues and merge these changes in a meaningful way.

When users initialise the ITS within a projects Git repository, a new orphan branch is created to act as the "issue repository". An orphan branch is a branch that has no commit history and is totally disconnected from all other branches. All data related to the ITS is stored on this branch. This is so that issue data is not mixed in with the files of the original project.

Individual issues are stored on the top-level of this branch as can be seen in figure 9. Each issue's data is stored in a uniquely named directory. This unique name is the SHA256 hash value of the issue's original title, description, creation time, and creator (name and email). Also found in the top-level of the issue branch is the ".git" directory and a file ".nicknames". The ".nicknames" file is a JSON file containing a map of nicknames for issues. The reason the directory name needs to be unique is so that Git will not try to incorrectly merge two issues during synchronisation.

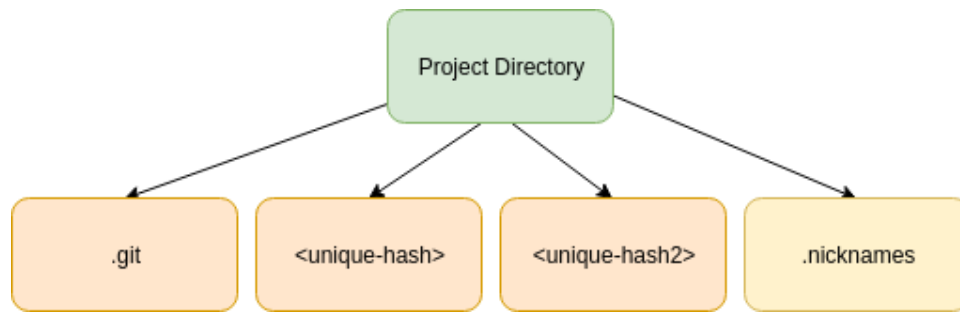


Figure 9 - Issue Repository Top Level Directory

Inside the directory of an issue, two more directories and several files are contained as seen in figure 10. One directory is for comments stored on the issue and the other is for media attachments such as pdfs or images. The files contained in this directory are for storing the issue's attributes e.g. title and description.

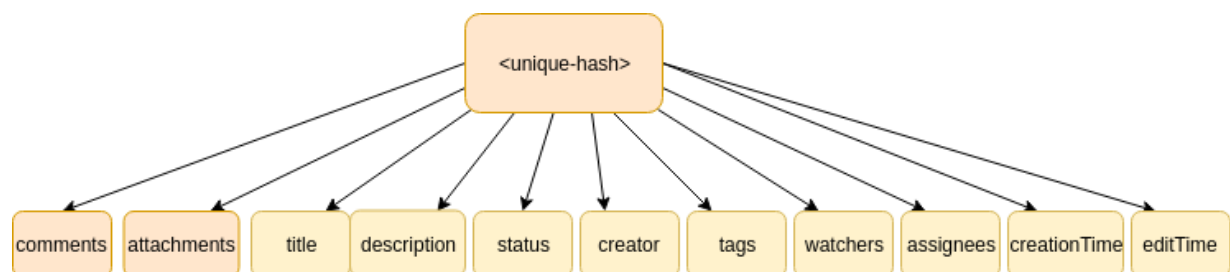


Figure 10 - Issue File Structure

There are multiple reasons why each attribute has its own file. The main reason is because Git is extremely efficient at storing small files text files. In short, Git makes a snapshot of the content of these files and stores a reference to it. If two issues in the repository have the same content, Git will not store two copies, it will just store another reference to the content. This is useful as it will be common for multiple different issues to contain duplicate data such as having a status of "open" or a tag "bug". Using individual files also means that content inside is easier to manipulate by the system when issue data is updated. If issue data was all in one file, much more parsing would need to be done to get the required field.

As mentioned above, comments are stored in a sub-directory of their related issue. The file structure for comments replicates the structure used for storing issues. Each comment is stored under a directory with a unique name. The directories name is the SHA256 hash value of the comments author, message, and creation time. The data contained in each comment directory is also in individual files for the same reason as discussed above.

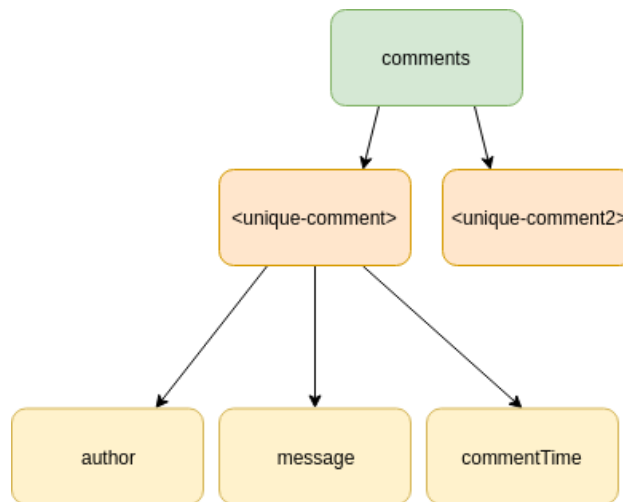


Figure 11- Comment File Structure

The other directory stored in an issue is for attachments. Attachments are stored on the top level of the attachments directory as shown in figure 12. Attachments require a unique name or else they are overwritten (e.g. if a file “example.txt” exists and a user attaches another “example.txt” the first file will be replaced; however, the old will still exist in the issue repo’s commit history). As Git can bottleneck (when synchronising) with large binary files, attachment size is capped at 20MB. However, if more time was available Git-annex (Git Annex, 2019) would be used to support large binary files. This is discussed more in chapter 8.2.5.

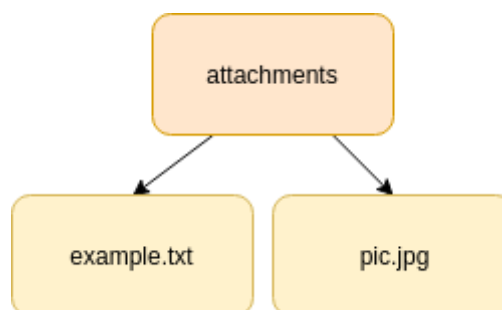


Figure 12- Attachments File Structure

The data contained in the files of issues and comments is in JSON format. The reason JSON is used is because it is simple to be parsed by the system and for users to read should they have to resolve a merge conflict. Users can also choose to manually edit the files and backup issues using whatever tool they prefer. Originally issues were stored as serialised Java objects. However, this was not practical for merging as it gave no flexibility. An example of this is if there were two conflicts and the user wanted to use the remotes change for one attribute and keep their own local change for the other it would be impossible. They would have to choose between using the remotes changes for both conflicts or their own.

4.4 Distributed Issue Tracking System

The ITS is made distributed because it extends the features that make Git distributed. When issues are synchronised between repositories, it does a full clone of the issue repository. This means that a full history of every issue is included. Updates to the issue repository are distinguished by a new commit. Each commit contains a snapshot of the issue repository at that time and tracks who made the update. This provides the ability for issue's to be reverted to an early version if needed and captures who made the update. As every user has a full copy of the issue repository, this serves as multiple backups and if one users' repository is lost or corrupt then they can just re-clone it. There is no single point of failure like in a centralised system.

Chapter 5 – Detailed Design and Implementation

5.1 Implementation Language

The main requirement for any language used in this project is that it must be able to interact with Git. Any language that can use the running operating systems terminal could interact with Git directly. However, the decision was made to use a library providing Git functionality as this would be easier to work with, more portable and easier to test. There are several programming languages this project could have used. In this section a small comparison of the languages considered is done. This is to highlight the strengths and weaknesses of their suitability for this project.

5.1.1 Java

Java is a high-level, general-purpose programming language with a focus on the object-orientated programming paradigm. This allows for the creation of modular and reusable code. An advantage of using Java is that it is the language the developer of the system has the most experience with. During university, several classes used this language to teach object-orientated design and concepts.

Java can be used with Maven which allows for easy use of external libraries and packaging of the project. To interact with Git there is a library called JGit (Eclipse, 2019). This is a library recommended by the official Git website (Git, 2019) for any Java projects needing to use Git. The library provides a high-level API which replicates basic Git commands e.g. “git push” and a low-level API which allows developers to work directly with the objects that make up Git repositories. As Java is platform independent due to the Java virtual machine (JVM) and JGit is written in native Java, the system will be portable. This means that the system can run on any OS that has the JVM. The main disadvantage of using Java is the start-up time for the JVM. The system does not stay running in the background, so for every command the user enters: the JVM is started; the required classes loaded, the system run; and then exits. This takes significantly longer to do than in a language like C/C++ which runs directly on the computer’s processor as native code.

5.1.2 Python

Python is a high-level, interpreted, general-purpose programming language. There is no main programming paradigm associated with Python. It is commonly classified as functional, imperative and object-orientated. Python provides a comprehensive standard library as well as easy access to external libraries. The ability to interact with Git is provided using the GitPython library (GitPython, 2019). This library provides both a high- and low-level API with the same capabilities of the JGit library referenced above. One of the requirements for GitPython is that Git needs to be installed. Due to this requirement

Python is less portable when compared to Java for this project as the JGit library has no dependencies. One of the main benefits of Python over Java is the start-up time. In a benchmark for start-up times of different languages (Drung, n.d), Python proves to be much faster than Java. This can save end users a significant amount when running the system repeatedly. The main disadvantage for using Python for this project is the developers limited experience. The developer has only used Python in one class during university so a large amount of time spent on the project would be learning how to use the language.

5.1.3 Go

Go is a statically typed, compiled programming language. As with Python, Go can be classified into multiple paradigms such as concurrent, functional, imperative and object-orientated. Go has multiple options for Git functionality: Go-git (source[d], 2019) and libgit2/git2go (libgit2, 2019). Both libraries have extensive Git functionality equal to JGit and GitPython, additionally both libraries have been written with portability in mind. Go-git is written in pure Go so can be run from anywhere Go is installed. Git2go is a Go binding on libgit2. Libgit2 is an ANSI C implementation of the core methods in Git. In the survey of start-up times, Go is significantly faster than both Java and Python. The disadvantage of using Go in this project is that the developer has minimal experience with the language. Therefore, a significant amount of the project's time would need to be spent on learning the language.

5.1.4 Chosen Language

Java was the language chosen for the project. The reason for this is due to the developer's experience the language and the portability of the project to other operating systems. Go seems like the optimal choice due to the speed and portability of the language. However, the lack of experience in the language was likely to create a large constraint for the project due to spending time learning about the language.

5.2 Libraries Used

5.2.1 JGit

JGit (Eclipse, 2019) is a library that provides extensive Git functionality written in native Java. There are two levels to the API: porcelain (high level) and plumbing (low level). The terminology of these come from Git (Git, 2019). The porcelain API provides a simple way for basic Git commands to be implemented into the code. An example command in the porcelain API is git push. Using the porcelain API, a file can be pushed in one line of code. The plumbing API provides a way to interact with the low-level objects in a Git repository directly. However, the code is much more verbose and complex than the porcelain API. The usage of this API for the project is explained later in this chapter of the report.

5.2.2 Commons CLI

Commons CLI (Apache, 2019) is an Apache library for handling input passed in from a command line interface. It is used in the project to ensure that the required options for a command are specified. It handles almost all of the parsing and provides a simple API to check if the correct arguments are present.

5.2.3 Jackson

Jackson (FasterXML, 2019) is a Java library used for parsing or generating JSON. It is used in the project when writing the attributes of an issue to a file and when parsing the JSON in the files back into an Issue object.

5.2.4 OpenCSV

OpenCSV (OpenCSV, 2019) is a Java library that is used for reading and writing CSV files. It is used by the issue tracking system to export issues and their comments.

5.2.5 Commons IO

Commons IO (Apache, 2019) is an Apache utility library that assists with IO functionality. It is used throughout the project for tasks such as creating or deleting temporary directories and reading the content of files.

5.3 Development Tools

5.3.1 Maven

Apache Maven (Apache, 2019) is a tool used for building and managing Java based projects. Maven projects use a standardised file structure. Normal source code for a project is stored under the src/main directory and tests are stored in a separate parallel source tree under the src/test directory. Maven provides a standardised way to build projects using its project object model (POM). Here dependencies for the project can be specified. Example dependencies are external libraries e.g. JGit. The dependencies will be packaged with the projects code.

Using Maven allowed for the libraries outlined in the previous section to be packaged into the JAR file used to run the project. This meant that users of the JAR do not have to manually add several libraries to their Java class path.

5.3.2 IntelliJ IDEA Ultimate

IntelliJ IDEA (JetBrains, 2019) is the IDE that all development of the project was done in. IntelliJ is a popular IDE that provides several useful tools to improve productivity such as code completion, code quality suggestions, refactoring tools and a debugger.

5.4 Strategy Design Pattern

The strategy design pattern was used in the system to handle how the system executes commands. The strategy pattern allows an algorithm to be selected for use at runtime. This is necessary because the system does not know which command the user is going to use until then. Using a common interface, a set of algorithms were created for the system to choose at run time. The structure of the strategy pattern is:

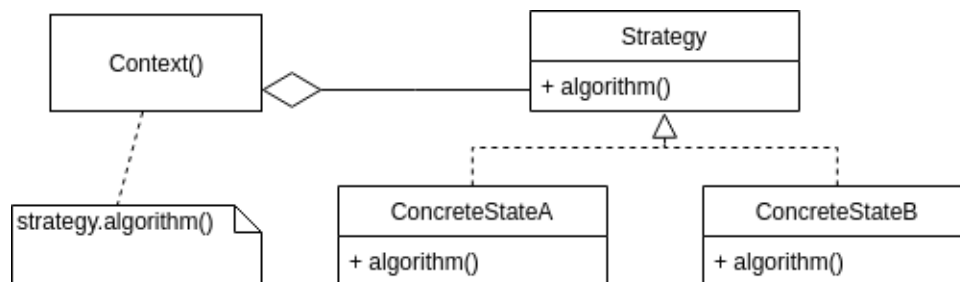


Figure 13- Strategy Design Pattern

Using the strategy pattern avoids having multiple conditionals. The configuration is done in one place for the whole program. In the project this is done in the **CommandSelector** class. The command (strategy) to execute is then passed into the context (**CommandExecutor**). This is adhering to the open-closed SOLID principle. The class is closed for change regarding how to solve the task (calling `execute` method), but the design is open to adding more algorithms (Commands) to solve the task. Using classes from the project the structure of the pattern is:

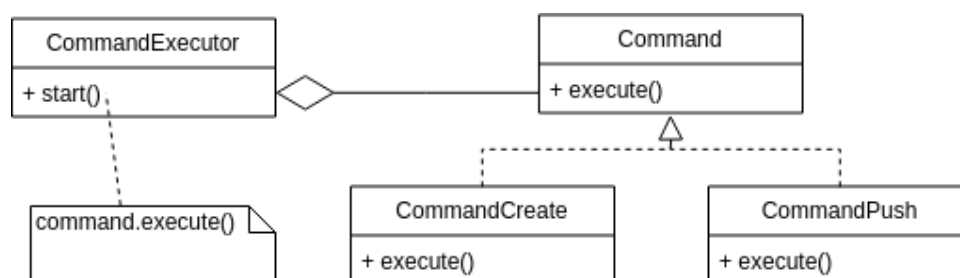


Figure 14 - Strategy Design Pattern - Project Context

The full class diagram of the system and an explanation of the design can be found in appendix D.

5.5 Challenging/Interesting Aspects of Implementation

5.5.1 Storing Issues

The most challenging part of the implementation was working with the internals of Git using the JGit plumbing API. The reason the low-level API was used is because the porcelain API required a branch to be checked out in order to write the issues. Requiring a user to check out the issue branch would mean that they need to commit their working directory's changes every time they interact with the issue tracking system. This would make the ITS undesirable for developers to use because sometimes they will need to commit incomplete code. Knowledge of the internals of Git is required to commit files to an issue branch without a Git working directory.

The underlying foundation of Git is a graph structure. Git is a content-addressable filesystem meaning that it is essentially key-value data storage (Git, 2019). The values that are stored are called objects. There are multiple types of objects and all of them are stored inside a repository's ".git/objects" directory. In native Git, content such as a file is stored by using the command "hash-object". This stores the data and returns the unique key that can be used to find the object inside the repository. In this project the following method is used to replicate the functionality of "hash-object":

```
ObjectId insert(int type, byte[] content) throws IOException {
    ObjectInserter objectInserter = repository.newObjectInserter();
    ObjectId result = objectInserter.insert(type, content);
    objectInserter.flush();
    return result;
}
```

Figure 15- Object Inserter - JGit Version of git-hash

In this code snippet there are a few key things to know about. The parameter "type" is referring to the type of object being inserted into the repository. As previously stated, there is multiple types of objects in Git. These are blobs, trees and commits. The "content" parameter is the actual data that is being stored in the repository. Returned from this method is an ObjectId which is the key to the content stored. The use of this ObjectId is described later in this section.

The "blob" object type is what is created when a file is inserted into the repository. The blob consists only of the content inside the file not its filename. If two files with different names contain the same content, then only one copy of the data is stored and the same ObjectId is returned.

The “tree” object type can be seen as a simple file system. The tree object is what will store the directory structure and filenames for a group of files. Tree objects can contain blobs (files) and other tree objects (sub-directories).

The tree object for an issue of this project can be visualised as the following figure:

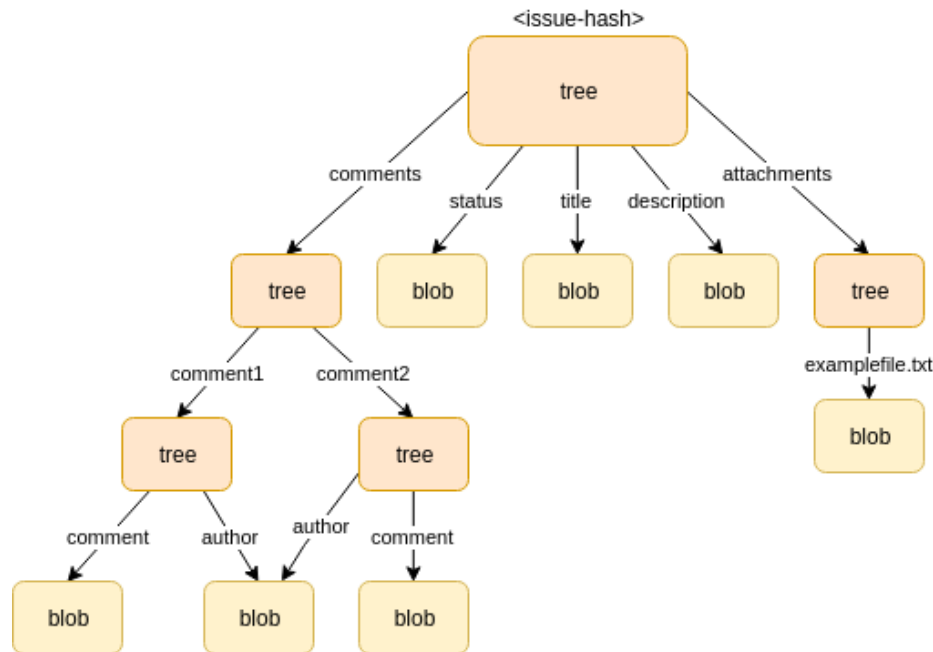


Figure 16- Issue Tree Object

Some attributes of an issue (e.g. tags) are not shown as they are just more blobs on the 2nd level of the tree. An example of two files containing the same content is shown in the comment trees. As both comments have the same author and the contents are the same, the tree will point to the same blob.

The issue tree is built manually in the project. To following code snippet shows the process of building the tree object for an issue. The code is simplified for two attributes of an issue (title blob and comment tree). Since there is no working directory, when an issue is being written to the issue branch it converts its attributes from their Java type to a blob instead of the normal Git process of converting a file to a blob.

```

ObjectId buildIssueTree(Issue issue) throws IOException {
    TreeFormatter issueTreeFormatter = new TreeFormatter();

    byte[] titleJson = mapper.writeValueAsBytes(issue.getTitle());

    ObjectId comments = buildCommentTree(issue.getComments());
    ObjectId title = oi.insert(Constants.OBJ_BLOB, titleJson);

    issueTreeFormatter.append( name: "comments", FileMode.TREE, comments);
    issueTreeFormatter.append( name: "title", FileMode.REGULAR_FILE, title);

    return oi.insert(Constants.OBJ_TREE, issueTreeFormatter.toByteArray());
}
  
```

Figure 17- Issue Tree Builder Method

The snippet shows a TreeFormatter object being created. This is used to help build the issue tree. Next the title attribute is converted to a byte array ready to be inserted to a Git repository. Following this the reference to the comment tree is obtained by calling a buildCommentTree method. This method is the same as the buildIssueTree method but the tree it builds contains the blobs for comment attributes. The reference to the title blob is obtained by using the insert method shown earlier in this section. These references are then appended into the TreeFormatter specifying what type of object they are, and what their file/directory name should be. The TreeFormatter is then converted to a byte array and inserted into the repository as a tree object. Finally, the reference to the issue's tree is returned.

The method the issue tree reference is returned to is the method that builds the tree for the full issue repository. It will call the buildIssueTree method for every issue stored and then insert the issues into a TreeFormatter. This TreeFormatter is then inserted into the repository and used in the last kind of Git object, a commit.

The "commit" object is essentially a snapshot of the project's files at a given time. It contains a tree object for reference to the project files and some metadata. The metadata contained in a commit is as follows: the user defined message; the creators name; the time of creation; and the parents of the commit. The parents of a commit are other commit objects representing a previous "snapshot" of the project.

```
CommitBuilder commitBuilder = new CommitBuilder();
commitBuilder.setMessage(message);
PersonId personId = new PersonId(Util.getUserName(repo), Util.getEmail(repo));
commitBuilder.setAuthor(personId);
commitBuilder.setCommitter(personId);

commitBuilder.setTreeId(tree);
commitBuilder.setParentIds(parent);
ObjectId newCommit = oi.insert(commitBuilder);
```

Figure 18- Commit Builder Method

The CommitBuilder object from JGit is used to create Git commit objects. In the code snippet, the tree ID and metadata of the commit is being set. This tree ID is the reference to the tree object created earlier containing all the issues of the repository. The CommitBuilder is then inserted to the repository providing a reference to the commit object. This reference is used to update the issue branch.

A branch in Git is simply a pointer to a snapshot of the project – a commit. Users can check out a branch and this will display the snapshot of the project files. Issues in this project's ITS are stored on their own branch. Therefore, when changes have been made to issues contained in the repository, the commit the issue branch refers to needs to be updated. This is done with the following code:

```
RevCommit commit;

try (RevWalk walk = new RevWalk(repo)) {
    commit = walk.parseCommit(newCommit);
    walk.dispose();
}

RefUpdate update = repo.updateRef("refs/heads/issues");
update.setNewObjectId(commit);
RefUpdate.Result result = update.update();

switch (result) {
    case NEW:
    case FAST_FORWARD:
        return true;
    default:
        return false;
}
```

Figure 19 - Update Issue Branch Ref

This code is immediately after the CommitBuilder code and shows the ref “refs/heads/issues” which is the issue branch being updated to point at the new commit. The term ref is used for anything pointing to a commit. If the branch is successfully updated, then the method will return true.

5.5.2 Pulling Issues

The second most challenging part of the project's implementation was pulling issues from a remote repository that would cause merge conflicts. It was decided that the system should let the user resolve the conflicts as there is no way for the system to decide which data the user wants to keep.

The pull command in Git is a Porcelain command that involves a combination of two other Git commands: fetch and merge. As the system is not able to check out the issue branch, another constraint was introduced to pulling issues. To implement the pull functionality several steps were taken. A clone of the local issue repository was made in a temporary directory. This is just a shallow clone as there is no need for the full repository history. From this clone, a fetch of the remote repository is done and then a merge is attempted of the remote and clone. If the system detects no merge conflicts, then a merge between the clone and local repository is done updating the issue branch. If the system detects any merge conflicts, then the option to resolve them is presented to the user. If the user chooses to resolve them then the default editor for the user's operating system is opened containing the merge conflicts. Once the conflicts are resolved, they are committed to the cloned repository. The local repository then merges with the clone using a "YOURS" merge strategy. A YOURS merge is using the clone's changes in any conflict; this is because the conflicts have already been resolved on the clone.

5.5.3 Query Language

The query language was an interesting feature to implement as there were several ways to approach defining the structure of the queries and the method of filtering issues. The queries were used to filter what issues are displayed when the user runs the ls command. The structure of the query is based on the queries in Issues (GitHub), this is because in the survey of issue tracking systems (JetBrains, 2016) users preferred how GitHub implemented the query language. The format of the queries in the system are "qualifier:value". The qualifier is the attribute of an issue that the user wants to filter by e.g. title or description. The value is what the user wants the issue's attribute to contain. Values with whitespace in them must be surrounded by quotation marks. Several queries can be chained together by separating them with a space e.g. "qualifier:value qualifier:value".

Unlike the systems CLI, the parser of the queries is implemented manually rather than using an external library. The individual queries are obtained by splitting the input on a space. This gives a list of the individual queries which are then split on the colon giving the qualifier and value. These are put into a HashMap and passed to the class which then filters the issues. When queries are being parsed, if they are in an invalid format then they are ignored and a message to the user is output telling which part of the query is invalid.

The method used to filter issues is simple. All issues (which are stored as a List) are streamed through and if they don't meet the conditions contained in the qualifier:value hashmap then they are removed. A method in the filter class looks at the qualifier and will use the appropriate helper method to check if the issue matches the criteria. An example of these helper methods can be seen below.

```
private boolean commentContains(Issue issue, String value) {  
    return issue.getComments().stream().anyMatch(comment  
        -> comment.getMessage().toLowerCase().contains(value));  
}
```

Figure 20 - Search Comment Message for String

Here an issue's comments are streamed through to check if they contain a certain value. If the value is not found in any comment, then the issue is removed from the list of issues. Once all issues have been checked, the list of filtered issues is returned. The filtered issues can be output to users in the terminal or used to filter the issues the user wants to export in a CSV.

5.6 Problems Encountered

5.6.1 Dirty Work Tree

The largest problem encountered during the project happened when implementing the code to store issues (implementation details above). The problem that occurred was the working tree was becoming dirty if the issue branch was checked out after issues were inserted to the branch. A dirty work tree is when file changes are not staged or committed inside the repository. When running `git-status` inside the repository, it would show that issues have been deleted and modified but the changes were not staged. Several days were spent trying to debug this issue and the documentation for JGit's plumbing API is not detail how to use it.

Several attempts were made to resolve this issue. The first was simply trying to do a hard reset of the repository after writing the issues. A hard reset is undoing all local changes to the repository, reverting the staging index and working directory back to the most recent commit. This did not work, and the status of the repository was still showing modified and deleted files. This highlighted that something must be wrong with the tree object contained in the commit.

The second attempted solution was building the tree object of the new commit out of the old directory cache and adding/removing the new/old issue trees contained. This approach failed as it proved extremely difficult to remove specific sub-trees from the cached trees.

The actual solution turned out to be much simpler than expected. The problem was solved once it was discovered that blob names must be added in alphabetical order when building the tree object. The solution was found reading a Stack Overflow post on the order of trees (Iscla, 2015).

5.6.2 Formatting the CSV files

To export issues from the system, the decision was made to store them as a CSV file. This is due to it being easy to use CSV data in other programs should the user wish to do so. The OpenCSV library was used to create the files. The first problem encountered was when naming and formatting the order of columns. By default, the library let you add names to the columns. These columns would then be stored in alphabetical order. This works but if a user was to open the CSV file e.g. in Microsoft Excel, it would show the assignees of an issue before the main attributes such as: title, description and status. The only other default option the library allowed was to manually choose the order to store data but did not provide a way to name the columns. The solution to this problem was to manually choose the order to store data and extend one of the classes in the library. This class is responsible for mapping data to columns, the method that generates column names was overridden and the names of the columns were hard-coded in. This can be seen in the snippet below:

```
private static class CustomMappingStrategy<T> extends ColumnPositionMappingStrategy<T> {  
    private static final String[] HEADER = new String[]{"HASH", "AUTHOR", "MESSAGE", "CREATIONTIME"};  
  
    @Override  
    public String[] generateHeader(T bean) { return HEADER; }  
}
```

Figure 21- Custom CSV Mapping Strategy Class

The next problem occurred when evaluating the data contained in the CSV. The first implementation of the CSV exporter only output one CSV file containing all attributes of an issue – including comments. This comment column was extremely hard to read and would be hard to use in another program. The solution to this problem was to create two CSV files. One with the main issue data and one containing the comments with a link back to the original issue's hash. The user could then optionally select if they wanted to include comments or not.

5.6.3 Nicknames

Issue nicknames were introduced to solve a large usability problem the system had when identifying an issue. Before adding the nickname feature, issues had to be referenced by their unique hash value (explanation of why in section 4.3). It is extremely unlikely a user will be able to remember a SHA256 hash value and specifying the issue they want to work with meant they would have to copy/paste the hash every time. To solve this issue, nicknames were introduced. A nickname is a unique name the user can choose to identify an issue instead of using the hash value. All nicknames are verified to be unique and an issue can have only one nickname. Nicknames can also be updated or removed. A method to find an issue by hash or nickname was then created as shown below.

```
public Optional<Issue> getIssue(String identifier) {  
    return getIssues().stream()  
        .filter((issue) -> issue.getHash().equals(identifier) || issue.getHash().equals(getHash(identifier)))  
        .reduce((u, v) -> {  
            throw new IllegalStateException("More than one issue with same hash");  
        });  
}
```

Figure 22- Find Issue Object with hash or nickname

Chapter 6 – Verification and Validation

In this chapter, the verification and validation procedures taken during the project to ensure the final product satisfies its specification are described.

6.1 Verification

The following table is used to verify that the system matches the specification. All requirements defined in the specification (section 3.3) are listed with an explanation of how the system meets this requirement.

Requirement	How Requirement is Met
Initialise an issue repository inside a pre-existing Git repository	Using the “init” command. This command creates a branch inside the repository that is used for issues.
Create an issue with a title, description, and a default open status	Using the “new” command. Users must specify the title and description. This creates an issue with an open status.
Close issues	Using the “edit” command. Users can give the CLI option “close”. This will change the issue status to close.
Reopen closed issues	Using the “edit” command. Users can give the CLI option “open”. This will reopen an issue with a closed status.
Add multiple tags to issues	Using the “tag” command with the add CLI option. Users can add multiple tags at once by comma separating them.
Remove tags from issues	Using the “tag” command with the “rm” CLI option. Users can remove multiple tags at once by comma separating them.
Add assignees to issues (assignees are users who must solve the issue)	Using the “assignee” command with the add option. This will add an assignee to an issue.
Remove assignees from issues	Using the “assignee” command with the rm option. Users can add an assignee to an issue.
Add watchers to issues (watchers are users who will receive notifications about updates to the issue)	Using the “watcher” command with the add option. This will add watchers to an issue.
Remove watchers from issues	Using the “watcher” command with the rm option. This will remove watchers from an issue.
Add media attachments to issues (e.g. pdfs and images)	Using the “attach” command with the add option. This will add a file to an issue’s contents.
Remove media attachments	Using the “attach” command with the remove option. This will allow a user to remove a media attachment from an issue.
Edit the title of an issue	Using the “edit” command specifying the title option. This will update the title of an issue with the given value.
Edit the description of an issue	Using the “edit” command specifying the description option. This will update the description of an issue with the given value.
Comment on issues	Using the “comment” command. This will allow a user to attach a comment to an issue.
View the full details of an issue	Using the “show” command. This will output all data contained in an issue.
View comments on issues	Using the “show” command. The show command shows all data in an issue including comments.
List all issues – displaying the id and title of all issues	Using the “ls” command. This will by default list all the issues in the repository showing the ID and title of every issue.
List all issues with a verbose option displaying all the information about the issue	Using the “ls” command with the “-v” option. This will display all information contained on an issue except for the comments.

Synchronise their repositories with others using push and pull commands	Using the “push” command will push issues to other repositories. Using the “pull” command will fetch and merge issues from a remote repo.
Issues are stored on their own branch inside a Git Repository	Issues are stored on a branch called “issues”. This branch is created by the “Init” command.
Issues track extra attributes like creation time, last edit time and who created the issue originally	This data is stored and can be viewed with the “show” command.
In the event of a merge conflict when synchronising repositories, users are presented with a way to fix the merge or abandon it	In the event of a merge conflict, the user is notified. If the user wishes to abandon the merge, they have the option of doing so and the repository is reset. If they chose to resolve it, the merge conflict is opened in the OS’s default editor.
System is distributed like Git – does a clone of the entire repository (full copy of issue history included)	When a project is cloned all issues and their history is contained. This is due to an extension of Git’s functionality.
System is decentralised – No need for a central server to sync, independent repositories can clone and synchronise with each other	The project’s ITS can push and pull with any remote. This requirement is evaluated in depth in the evaluation chapter of the report.
Users can filter which issues they see by using a query language	A query language was created and can be used with the “ls” command when specifying the “--query” option.
Query language can filter issues by title, description, status, tags, watchers, assignees, creators, edit time and creation time	The query language can filter issues by all these attributes.
Create CSV files containing information on issues	Using the “-csv” option with the ls command, all issues listed will be output to a csv file.
Create CSV files containing information on comments	Using the “-csvc” option with the ls command, all issues listed will be output to a csv file.
Data contained in CSV files can be filtered using the query language	Using a query in conjunction with the “-csv” or “-csvc” option will create a CSV with only the issues matching the query
Users can add a unique nickname to issues (makes the issue easier to identify in terminal)	Using the “nickname” command with the add option, users can specify a nickname for an issue.
Users can update an issue’s nickname	Using the “nickname” command with the update option, users can update a pre-existing nickname.
Users can remove an issue’s nickname	Using the “nickname” command with the rm option, users can remove a nickname from an issue.

All requirements of the system were successfully carried out. The next section validates that each of these requirements function without error. This validation is done by performing various methods of testing.

6.2 Validation

To ensure every feature of the system works correctly, extensive testing was carried out. The types of testing done were white box and black box.

6.2.1 Black Box Testing

Black box testing is where the functionality of the system is tested without the tester knowing the internal workings of the system. Tests are done from an end user perspective e.g. entering the command into the terminal.

The tests cases were created by examining the system requirements and use-cases (Appendix B.3). The use-cases were useful when creating test scenarios because they specify the expected behaviour of the system in different situations (e.g. normal input and invalid input).

In total 118 different scenarios were created to test the behaviour of each system feature. As there is so many scenarios, they are not included in the report body or appendices. To read them refer to the file “blackBoxTests” in the submission files. An example black box test is shown below:

Command Under Test:	Ls – using a query and exporting a csv			
Description of test taking place:	Pre-conditions	Test Data:	Expected Output	Actual Output
Tests that the “ls” command lists only issues matching the query. Also tests that only issues matching the query are exported to the csv file.	At least 1 issue exists with a title containing “hello” and description containing world and at least 1 issue without these values	Normal: -csv /path --query title:hello desc:world	Outputs all issues with specified title and description to terminal and also writes them to a csv file	Outputs all issues with specified title and desc to terminal and also writes them to a csv file
Result:	Pass			

Each command is tested with different data types. These are “normal”, “extreme”, and “exceptional”. Normal test data is user input that is valid (i.e. contains all required flags for a command and is in the correct format). Extreme test data is user input such as specifying an issue that does not exist to comment on. Exceptional test data is user input that contains invalid options. Using different data types allows us to validate the system behaviour for all usage scenarios.

6.2.2 White Box Testing

White box testing validates the internal workings of the code. Tests are done by the developer with full knowledge of the source code. In the project, the white box testing is done with Junit tests. Several unit tests were carried out for every method in the project. This was to ensure that every execution path for the system functioned correctly. The final test coverage for the system was 99%.

As line coverage on its own is not a good measure of quality testing, mutation testing was carried out on the project's test suite of 360 unit tests. Mutation testing is where parts of the code are changed (mutated) and then the test suite is run. The reason for doing mutation testing is because test coverage only measures which code is executed, it does not confirm that the tests can successfully detect bugs (PiTest, 2019). Mutation testing is an effective measure of test quality because when the code is mutated it should produce different results when tested, therefore failing the tests. If the tests do not fail, then it highlights that the code may not be tested correctly (PiTest, 2019).

Detailed test results for the project can be found in appendix C.

6.2.3 Testing Technologies

The Junit tests done on the system use several technologies that help to improve the quality of each test.

6.2.3.1 Mockito

Mockito (Mockito, 2019) is a testing framework for Java that allows creation of “mock” objects. These mock objects are used by a class under test (CUT). The CUT is the Java class being tested. The project uses Mockito when testing the classes that implement the Command interface. The command classes use a mock issue repository object. This is because in the command tests there is no need to test the issue repository code. Pre-programmed responses for the issue repository object's methods are set up so that the command class can be tested on how it handles different situations.

6.2.3.2 Junit Pioneer

Junit Pioneer (JUnit Pioneer, 2019) is an extension pack to Junit. As it is still in early development, it only offers a limited number of features. The feature used from this library creates a temporary directory that exists for the life cycle of one test, and then is cleaned up afterwards. The reason temporary directories are used is because when testing the issue repository's functionality, a lot of JGit features are used. It is a general rule of mocking objects that you do not mock any objects you do not own (not implemented yourself). For this reason, normal Git repositories are created inside these temporary directories for testing the issue repository code.

6.2.3.3 Pi-test

Pi-test (PiTest, 2019) is a mutation testing library for Java. Pi-test automatically modifies the code in the system, re-runs the test suite and then generates a HTML report showing the results of the mutation testing. The full mutation report for the system can be found in the “mutation-report” directory of the submission files.

Chapter 7 – Results and Evaluation

7.1 Evaluation Criteria

When gathering requirements for the project it was made apparent that an issue tracking system has potentially unlimited scope. Issue tracking systems like Jira or Issues by GitHub are very complex with an extensive number of features. The features these systems offer are often not related to issue tracking but instead to project management. In the timeframe of this project it would be impossible to replicate all the features of these systems. Therefore, only the core features of an issue tracking system (identified in project specification) and a few extras such as the query language and issue exporter were implemented.

The main evaluation criteria for the system is to ensure that it is fit for purpose; how does it compare to other issue tracking systems and how can it be used.

7.2 System Comparison

In this section, the project is compared with the issue tracking features of Jira and Issues (GitHub). The project management features these systems offer are outside the scope of this project and are therefore ignored in this comparison. The purpose of this comparison is to evaluate the usability of the project. This ensures that the system contains all the necessary features of an issue tracking system.

Features	Project (DDIT)	Jira	Issues (GitHub)
Users can create issues	✓	✓	✓
Users can add/update an issue title	✓	✓	✓
Users can add/update an issue description	✓	✓	✓
Users can add/remove tags	✓	✓	✓
Users can open/close issues	✓	✓	✓
Users can delete issues	✓	✓	✓
Users can comment on issues	✓	✓	✓
Users can search for specific issues	✓	✓	✓
Users can view all issues	✓	✓	✓
Users can add/remove media attachments	✓	✓	✓
Users can add/remove assignees	✓	✓	✓
Users can add/remove watchers	✓	✓	✓
Users can export issues	✓	✓	✓

These features were identified as fundamental to issue tracking systems in the background research of ITSs (section 2.4). This project successfully implemented all features and therefore in terms of features is fit for purpose as an ITS.

7.3 System Performance

7.3.1 Benchmark Testing

To evaluate the system performance, the system underwent a series of benchmark tests to measure how well it performs under stress. In order to stress the system 7000 issues were created. Each test used a script to run one of the issue tracker's commands and timed how long it took to execute. From the execution time the performance of the system can be judged. An example of these benchmark test scripts can be seen below:

```
lsTime="$(time ( git ddit ls ) 2>&1 1>/dev/null )"
echo "$lsTime"
```

Figure 23 - Ls benchmark script

In a bid to ensure all tests were performed under the same conditions some precautions were taken. Firstly, the computer was restarted before each test. This ensures that no issue data is cached in the computer's memory as this would make the system faster for any test that could access this data. Secondly, each test was run ten times to account for any OS noise e.g. JVM garbage collection or OS checking for updates. This noise could affect the processing power that could be allocated to running the tests. Lastly, the computer was booted into console mode and left unused apart from the running test. This was to reduce the chance of other processes interfering with the test.

7.3.1.1 Benchmark 1

The first benchmark test was creating 7000 issues. This test was primarily for creating the issues to be used in the other benchmarks but was useful to see how long it would take if, for example, a user was importing issues from another system. This test took a significant amount of time; 344 minutes for all issues to be created, yielding an average of 2.9 seconds per issue. This is likely due to the start-up time of the JVM before every command. This could be considered unusable but creating 7000 issues at one time is not going to be a common use for the system.

7.3.1.2 Benchmark 2

The second benchmark test was using the "ddit ls" command in a repository with 7000 issues. A project containing several thousand issues is likely to be a common scenario for an issue tracking system. An example of this is large-scale, open source projects such as Angular (Angular, 2018). Angular uses GitHub's ITS and as of March 22nd 2019, has 2200 open issues and 15000 closed issues.

After 10 runs of the test, the average time to list all the issues took 5.6 seconds. This is not a huge amount of time for a user to wait but compared to doing a normal “ls” in a bash terminal it is slow (ls takes 0.081 seconds). The only advantage of using the ITS’s ls command is that the title of the issue is output and if the verbose option is specified all other attributes of the issue are displayed as well.

7.3.1.3 Benchmark 3

The third benchmark test was using the “ddit ls” command again and using the query language to filter down to retrieve only 1 specific issue. A common scenario is for a user to search for a specific issue using its title. This test was again done in a repository with 7000 issues. After 10 runs, the average time for execution of this command was also 5.6 seconds.

7.3.1.4 Benchmark 4

The fourth benchmark test was repeating test 2 with 100 issues. This was to evaluate how the system would perform in a project with a small number of issues. After 10 runs, the average execution time for this test was 0.63 seconds. Compared to the second benchmark this is much more usable and there is almost no delay for the user.

7.3.1.5 Benchmark 5

The final benchmark test was repeating test 3 with 100 issues. This was to evaluate how quickly the system will find a specific issue in a small issue repository. The average execution time after 10 runs was 0.65 seconds. As with test 4, using the system with a smaller issue repository is much more usable.

7.3.2 Benchmarking Conclusion

When observing the benchmark testing results, it is clear that the system performance is relatively poor in repositories with an extremely high issue count. While the system is still usable i.e. it doesn’t crash the computer, waiting 6 seconds to list issues is not ideal for an end user. The results of the benchmark tests for repositories with few issues were very good. Issues were listed in around half a second which is ideal performance for an end user.

In conclusion, the system is inefficient for large issue repositories but effective for small ones. This is likely due to the way the ITS reads the issues from backing storage. In the current implementation, all issue data is loaded into memory and then the issues commands are executed. An alternative implementation that could possibly be more efficient would be to only read the files of an issue that are specified in a query. If the data inside the file matches what is in the query, then the rest of the data in that issue directory can be read. Otherwise, the system can carry on to the next issue. A way of improving the list commands performance would be to limit the amount of issues output. The system could be modified to only display 100 issues by default with an additional flag to display more.

This would increase the performance of the system significantly. The evidence for this can be seen in benchmark tests 4 and 5.

A copy of the test scripts can be found in the “sourcecode/scripts” directory of the submission file.

7.4 Potential Usage of System

There are several ways in which this system can be used. In this section the system will be evaluated on how well it works in different situations.

7.4.1 Workflows

In this subsection the system will be appraised on how well it works in different workflows.

7.4.1.1 Centralised Workflow

Using a centralised workflow (section 2.3.1) is a common practice of development teams. As described in the related work chapter of the report, developers of a project will directly synchronise their changes using a shared central repository. The issue tracking system will work well in this situation as every developer can make issues and directly share them to the main repository where other developers can obtain them. Centralised workflows are more common on smaller development teams or in an office setting where all developers can be trusted to make changes to the central repository. Although not in all cases, this means that the amount of issues contained in the project is much likely to be less than what is contained in an open source project like Angular. For this reason, the systems performance will be efficient and most likely faster to use than using a centralised web service.

7.4.1.2 Decentralised Workflow

A main requirement for the project was making the issue tracking system distributed and decentralised. Therefore, it is expected to also work in a decentralised workflow (section 2.3.2). The ITS can support multiple remote repositories so developers can synchronise issues whenever is convenient for them.

Often large open source repositories will use a decentralised workflow. They use this because not every developer can be trusted to directly make changes on the central repository. The issue tracking system may not be so effective in this scenario due to large open source projects often containing several thousand issues. Not only does it take a long time to find issues (refer to benchmark section) but having several thousand issues will take up lots of backing storage meaning the project will take a long time to clone. During the benchmark tests 7000 issues were created containing only a small amount of data in the title, description and status attributes. Although not much data was stored for each issue, it used 1.5Gb of backing storage. This is where a centralised system is better than using a distributed system. A project like angular with around 17000 issues, filled with comments and media

attachments will be extremely large. Therefore, keeping all this data on a separate web server will be more efficient for developers rather than needing to clone all these issues along with the project source code.

Decentralised workflows are not limited to large scale projects and can also be used in smaller development teams. The system would work well in these situations. Using the integration manager workflow from the related work chapter as an example. The developers would create issues and push them to the integration manager. This would give the integration manager the chance to review the issue before pushing it into the shared repository. This could help improve the quality of the issues i.e. if the developer has not clearly specified what the bug is, the integration manager could reject the issue or update the details if they know what the problem is.

7.4.2 Other Systems

Due to the generalised implementation of the ITS, other programs can use it. An example usage could be a CRON job which highlights outstanding issues. It could do this by piping the output of the system using a query to display issues older than a certain date or it could export the issue data as a CSV and parse this. Another use could be reporting software. Again, this could pipe the output or use a CSV containing issue data and build HTML reports from issue data. This would be like the “Pulse” feature of GitHub that generates reports from issues.

7.5 Portability Evaluation

A requirement for the system was to run on Linux. As the system was developed and tested on Linux this requirement was easily met. It is common for developers on the same team to use different operating systems, so this section is dedicated to testing on Windows. It would be useful to evaluate portability to MacOS however due to time constraints only Windows is tested.

Every command of the system was tested, and all were executed successfully. The only problem encountered was during the pull command. Issues were successfully pulled from the remote into the local repository; however, the temporary directory the system uses as an intermediary while merging (the second part of the pull command) failed to be deleted. This may be due to Java not having permissions to delete files in the Windows temporary directory.

When teams of developers work on different operating systems and collaborate using Git, a common problem that comes up is line endings. Unix and Windows operating systems use a different way to specify a new line in a file. Unix uses “\n” and Windows uses “\r\n”. To test how the system handles collaboration between different operating systems, issues were created on Linux and pushed to a remote GitHub repository. These issues were then pulled from a Windows machine, opened in an

editor, edited, and then committed. This was to test if Windows would change the line endings. After these edits, git diff was used to show the difference between the old commit (commit from Linux) and the new (after edit). Git diff showed only the changes done in title and edited attribute files and no changes to the line endings. This means the system is portable and can be used between collaborators on different operating systems.

7.6 Evaluation Conclusion

After several kinds of evaluation, it is clear that creating the distributed and decentralised issue tracking system was a success. Although the performance was not exceptionally good for extremely large issue repositories, the system still functioned correctly. All core features of an issue tracking system were implemented and can be used in a variety of workflows. Extensive testing (section 6.2) was carried out confirming that all features are implemented correct. Due to these reasons, the system can be considered fit for purpose.

Chapter 8 – Summary and Conclusion

8.1 Project Summary

Over the course of the project a distributed and decentralised issue tracking system (ITS) was developed. Multiple issue trackers were researched to gather the main requirements for this type of system, and all identified were successfully implemented. In addition to the original requirements, some extra features were added to the system such as the query language and issue exporter. These features make the ITS more generalised and allow for it to be used with other programs.

Due to the system performance evaluation taking more time than expected, the project process evaluation had to be omitted from the report. However, in a brief summary the project went smoothly with no major problems encountered.

8.2 Future Work

As mentioned previously in the report, the scope of an issue tracking system is extremely wide. In this section additional features or processes which could improve the system are described.

8.2.1 Git Hooks

During the project's development Git hooks were discussed as a possible usage of the system. However, due to lack of support from the JGit library this idea was unable to be demonstrated. Git hooks are scripts that Git executes before or after events such as push or pull (Git Hooks, 2019). The hope was to use the ITS in a hook so that when a developer pulls issues into their repository, they receive a message if they have been assigned to any issues. This example hook was created and works if using the official "git pull" command but JGit does not yet support post-merge hooks so it will not work with the systems pull command. A copy of this script can be found in the submission files in "sourcecode/scripts/post-merge". When JGit adds support for more types of hooks then it would be useful to add the functionality that allows the system to execute them.

8.2.2 Signed Commits

Currently in the ITS there is no moderation over who modifies the issue repository. Users have full control over their own local Git repository and can make any change they want. In the current version of the system, when users make a commit to the issue repository they are identified using the ".gitconfig" file located in their home directory. This is easy to spoof therefore; the system would benefit from something that can reliably identify users. Signing commits using GPG would verify that the changes to the issue repository came from a specific person. This would be beneficial as users could be held accountable for their actions.

8.2.3 Survey with Expert Users

To better evaluate the usability of the system, it would have been valuable to conduct a survey with expert users of Git. The reason this evaluation was not done is because it would have been too difficult to obtain enough expert users of Git to make the survey worthwhile as a criterion of evaluation.

8.2.4 Invert Option in Query Language

A useful feature to add to the query language would be to add an option which allows users to invert the results of the query. An example of this would be if the query was “title:hello” then instead of only displaying issues with “hello” contained in the title, it would show only issues that don’t contain “hello” in the title. The benefit this brings to the system would be it gives users more flexibility when filtering issues.

8.2.5 Git Annex

Currently in the system, users are limited to storing attachments of 20Mb or less. This decision was made because Git is not the most efficient at storing binary files. A useful addition to the system would be support for Git-annex (Git Annex, 2019). Git-annex “manages files with Git, without checking the file contents into git” (Git Annex, 2019). The benefits of this would be that any size of attachments could be added to the issues and users would not have to wait a long time to synchronise issue repositories. Git-annex will do its own separate synchronisation of issue attachments.

8.3 Conclusion

In conclusion, the project was a success. The objectives identified for the project were accomplished and the resulting system was evaluated to be fit for purpose. I believe that the design and implementation are of high standard and that the testing done of the system is rigorous and high quality. Overall, I am pleased with what I have achieved and have learned a lot.

Appendix A – References

Angular, 2018. *Ivy Renderer*. [Online]

Available at: <https://github.com/angular/angular/issues/21706>

[Accessed 09 March 2019].

Apache, 2019. *Commons CLI*. [Online]

Available at: <https://commons.apache.org/proper/commons-cli/index.html>

[Accessed 17 March 2019].

Apache, 2019. *Commons IO*. [Online]

Available at: <https://commons.apache.org/proper/commons-io/>

[Accessed 17 March 2019].

Apache, 2019. *Maven*. [Online]

Available at: <https://maven.apache.org/>

[Accessed 17 March 2019].

Apache, 2019. *Subversion*. [Online]

Available at: <https://subversion.apache.org/>

[Accessed 21 March 2019].

Atlassian, 2019. *Jira*. [Online]

Available at: <https://www.atlassian.com/software/jira>

[Accessed 21 March 2019].

Atlassian, 2019. *Jira Issue*. [Online]

Available at:

<https://confluence.atlassian.com/jirasoftwarecloud/files/941618698/966684803/1/1550814075606/Full+page+issue.png>

[Accessed 8 March 2019].

Deisz, K., 2016. *Exploring CLI Best Practices*. [Online]

Available at: <https://eng.localytics.com/exploring-cli-best-practices/>

[Accessed 13 March 2019].

Dropbox, 2019. *Dropbox*. [Online]

Available at: <https://www.dropbox.com/>

[Accessed 21 March 2019].

Drung, B., n.d. *Startup-time*. [Online]

Available at: <https://github.com/bdrung/startup-time>

[Accessed 17 March 2019].

Eclipse, 2019. *JUnit*. [Online]

Available at: <https://www.eclipse.org/junit/>

[Accessed 17 March 2019].

FasterXML, 2019. *Jackson*. [Online]

Available at: <https://github.com/FasterXML/jackson>

[Accessed 17 March 2019].

Fowler, M., 2003. Use Cases. In: *UML Distilled (Third Edition)*. s.l.:Addison-Wesley Professional, p. Chapter 3.

Ganz, J. & Beyer, M., 2019. *git-dit*. [Online]
Available at: <https://github.com/neithernut/git-dit>
[Accessed 15 January 2019].

Git Annex, 2019. *Git Annex*. [Online]
Available at: <https://git-annex.branchable.com/>
[Accessed 24 March 2019].

Git Hooks, 2019. *Git Hooks*. [Online]
Available at: <https://githooks.com/>
[Accessed 24 March 2019].

Git, 2019. *About*. [Online]
Available at: <https://git-scm.com/about/distributed>
[Accessed 09 January 2019].

Git, 2019. *Distributed Git - Distributed Workflows*. [Online]
Available at: <https://git-scm.com/book/en/v1/Distributed-Git-Distributed-Workflows>
[Accessed 21 March 2019].

Git, 2019. *Embedding Git in your Applications - JGit*. [Online]
Available at: <https://git-scm.com/book/en/v2/Appendix-B%3A-Embedding-Git-in-your-Applications-JGit>
[Accessed 17th March 2019].

Git, 2019. *Git Internals - Git Objects*. [Online]
Available at: <https://git-scm.com/book/en/v1/Git-Internals-Git-Objects>
[Accessed 18 March 2019].

Git, 2019. *Git-notes*. [Online]
Available at: <https://git-scm.com/docs/git-notes>
[Accessed 10 January 2019].

GitHub, 2019. *GitHub*. [Online]
Available at: <https://github.com/>
[Accessed 21 March 2019].

GitLab, 2019. *About*. [Online]
Available at: <https://about.gitlab.com/>
[Accessed 24th March 2019].

GitPython, 2019. *GitPython Documentation*. [Online]
Available at: <http://gitpython.readthedocs.org/>
[Accessed 17 March 2019].

Google, 2019. *GitHub*. [Online]
Available at: <https://github.com/google/git-appraise>
[Accessed 10 January 2019].

IBM, 2019. *What is Distributed Computing*. [Online]
Available at:

https://www.ibm.com/support/knowledgecenter/SSAL2T_7.1.0/com.ibm.cics.tx.doc/concepts/c_wht_is_distsd_comptg.html#c_wht_is_distsd_comptg
[Accessed 08 January 2019].

Iscla, G., 2015. *Creating commit with Jgit and plumbing commands*. [Online]
Available at: <https://stackoverflow.com/questions/30261593/creating-commit-with-jgit-and-plumbing-commands/30262342#30262342>
[Accessed 20 March 2019].

JetBrains, 2016. *Tracking Tools Report 2016*. [Online]
Available at: <https://www.jetbrains.com/youtrack/promo/tracking-tools-report-2016/>
[Accessed 11 March 2019].

JetBrains, 2019. *IDEA*. [Online]
Available at: <https://www.jetbrains.com/idea/>
[Accessed 17 March 2019].

JUnit Pioneer, 2019. *JUnit Pioneer*. [Online]
Available at: <https://junit-pioneer.org>
[Accessed 24 March 2019].

libgit2, 2019. *git2go*. [Online]
Available at: <https://github.com/libgit2/git2go>
[Accessed 17 March 2019].

Mockito, 2019. *Mockito*. [Online]
Available at: <https://site.mockito.org/>
[Accessed 24 March 2019].

Muré, M., 2019. *git-bug*. [Online]
Available at: <https://github.com/MichaelMure/git-bug>
[Accessed 11 January 2019].

OpenCSV, 2019. *OpenCSV*. [Online]
Available at: <http://opencsv.sourceforge.net/>
[Accessed 17 March 2019].

PiTest, 2019. *PiTest*. [Online]
Available at: <http://pitest.org/>
[Accessed 24 March 2019].

source{d}, 2019. *go-git*. [Online]
Available at: <https://github.com/src-d/go-git>
[Accessed 17 March 2019].

Spinellis, D., 2019. *git-issue*. [Online]
Available at: <https://github.com/dspinellis/git-issue>
[Accessed 11 January 2019].

Appendix B – Detailed Specification and Design

B.1 Sprint details

B.1.1 Development Sprint 1 (14/01/2019 – 21/01/2019)

During this first week-long sprint, the first feature implemented was a command line interface supporting all the required commands for the system. Invalid flags were rejected and required options were enforced. The next feature to start implementing was where to store issues in the system. A new branch was created relatively easy by the system and for now this was sufficient to be the issue repository. Next the basic issue template was created. Issues at this point contained a unique id which consisted of their title, description, creation time and creators name hashed using a SHA256 function. To store the issues in the issue repository Java object serialization was used. The system would checkout the issue branch, write the issue and then re-checkout whatever branch the user was previously on.

When reflecting on this sprint instantly I was able to realise what was going to work and what wasn't. The basic issue format was fine however, doing a git checkout of the issue branch to write the issue and then returning to the original branch would not be appropriate as an end user. The reason being that if they had uncommitted changes, they would either have to lose their work or they would have to commit their working directory before using the system which is not practical. The next problem was with the CLI, although it functioned well and supported all the commands, when a user was inputting arguments the order was enforced. This is not a big problem, but it is seen as a usability issue and makes the cli harder to use than it needs to be.

B.1.2 Development Sprint 2 (22/01/2019 – 29/01/2019)

During this sprint I decided to continue with the original plan of creating more detailed issues and adding support for different commands. The reason for this was that the system was decoupled enough that I could write this functionality and change the problems from the first sprint at another time. This sprint went smoothly and reflecting upon it no changes were required.

B.1.3 Development Sprint 3 (30/01/2019 – 06/02/2019)

Sprint 3 was originally planned to be for implementing basic synchronisation of repositories however, due to the problems in the functionality in sprint 1 this sprint was dedicated to redoing the CLI and issue repository. Several different approaches were tried on how to store data including using git-submodules and git-worktree but none of these worked. The full details on how data storage works is in the design section of the report but in short, the solution works by manually updating references in

the Git repository. This was the hardest part of the project as git blobs had to be created and then trees had to be built from these blobs. Once the tree was built then a commit could be created and then references could be updated. The file type of the issues was also updated to be more user friendly and easier to work with. Issues were no longer serialised java objects; they were now individual JSON files for each field of an issue.

The next feature to improve this sprint was to redo the CLI, I rewrote the CLI two times and was still not happy with the result. The input arguments were no longer order enforced but the code was highly coupled. On the third attempt I managed to reuse the same design as the original CLI but made it so that args were fully flexible in order.

On reflection this sprint went extremely well. I set out to fix the problems of the first sprint and succeeded.

B.1.4 Development Sprint 4 (07/02/2019 – 10/02/2019)

During the fourth sprint basic synchronisation of repositories was to take place. Synching repositories can happen in two ways: pushing and pulling. I first set out to implement the push command as no difficult merging would have to take place. This was relatively simple to implement, if issues could be pushed with a fast-forward merge on the other repository, then it would do it automatically using the JGit library if not changes were rejected. Implementing the pull command was more challenging. Since I was not concerned about merge conflicts at this time, a fetch method was created and then using a YOURS merge strategy I merged the local and remote changes and then updated the issue repositories references.

This short sprint went as planned however I could tell that when merge conflicts occurred that it would be challenging to resolve.

B.1.5 Development Sprint 5 (11/02/2019 – 21/02/2019)

During this week the goal was to complete synchronisation of repositories. Since push was successfully implemented and pull half complete, I just had to handle merge conflicts. A few different strategies were tried, and it took less time than I thought to complete. Using a temporary directory, I cloned the local project and fetched the remote manually updating the refs of the cloned repository to include the merge and then done a merge of the cloned repo and the original local repo using a YOURS strategy. This took around 3 days to complete leaving time to do other required tasks of the project. During this time, I wrote the help texts for each command.

B.1.6 Development Sprint 6 (22/02/2019 – 28/02/2019)

The last sprint was added to implement a few extra requirements. During weekly progress meetings with my supervisor he suggested making some features to make the system usable by other programs. To do this I implemented a basic query language and the ability to export issues to CSVs. The query language was simple to implement, and the CSVs only required one small change after I wrote them and that was to put comments into a separate CSV and link them back to the original issue. Another requirement was to make issues easier to identify, up until this point issues were referred to by a SHA256 hash, this is not user friendly so I implemented a nickname feature so that users could then use a unique nickname for issues.

After this sprint I was happy that enough functionality was implemented and after some small code refactors to make the code more readable, I moved on to the testing phase.

B.2 Project Plan

This appendix contains the details of the original project plan and the updated project plan.

B.2.1 Original Project Plan

14th January to 11th of February - Implementation

- Development Sprint 1 (1 week): Create CLI, issue branch and basic issues
- Development Sprint 2 (1 week): Create detailed issues, view, comment on and update issues
- Development Sprint 3 (0.5 week): Basic synchronising with other repositories (ignoring merging issues)
- Development Sprint 4 (1.5 week): Complete synchronisation with other repos – all system functionality completed
- Documentation for each sprint's results

12th of February to 21st of February - Testing

- Create testing strategy: Black-box (unit testing), integration testing
- Fix any bugs highlighted from testing and document them
- Testing documentation (verification and validation section of report)

22nd of February to 22nd of March – Report Writing

- 1st week: Create report introductory pages, introduction section and finish problem description and specification section of report
- 2nd week: System design section of report, user and technical guide, installation guide, maintenance and developer guide
- 3rd week: Detailed design and implementation section of report
- 4th week: Results and evaluation section, summary and conclusion of project

23rd of March to 25th of March – Submission

- Report review: proof reading, formatting, polishing
- Submit project

27th of March - Demo

- Project Demonstration

B.2.2 Updated Project Plan

14th January to 28th of February - Implementation

- Development Sprint 1 (1 week): Create CLI, issue branch and basic issues
- Development Sprint 2 (1 week): Create detailed issues, view, comment on and update issues
- Development Sprint 3 (1 week): Reimplement CLI and data model for storing issues
- Development Sprint 4 (0.5 week): Basic synchronising with other repositories (ignoring merging issues)
- Development Sprint 5 (1.5 week): Complete synchronisation with other repos
- Development Sprint 6 (1 week): Implement a basic query language for filtering issues, add the ability to export issue data in CSV format
- Documentation for each sprint's results

1st of March to 7th of March - Testing

- Create testing strategy: Black-box (unit testing), integration testing, mutation testing
- Fix any bugs highlighted from testing and document them

8th of March to 23rd of March – Report Writing

- 8th March: Finish background research and related work chapter
- 9th-11th of March: Write problem description and specification chapter. Also write any appendices related to this chapter.
- 12th-13th of March: Write system design chapter and any related appendices
- 14th-15th of March: Write detailed design and implementation section of report and related appendices
- 16th-17th of March: Write verification and validation section of report
- 18th-21st of March: Write results and evaluation section of report
- 22nd-23rd of March: Write summary and conclusion for the project

24th & 25th of March – Submission

- Report review: proof reading, formatting, polishing
- Submit project

27th of March - Demo

- Project Demonstration

B.3 Use Cases

Title (goal):	Create Issue Repository
Actor(s):	Developer
Description (story):	Creates an issue repository within a Git repository. Developers can start to use the ITS e.g. create issues.
Pre-Conditions:	Current working directory is a Git repository.
Post-Conditions:	Issue branch is created.
Main Success Scenario:	<ol style="list-style-type: none">1. Dev types command to initialise issue repo2. Dev is notified that an issue repository is successfully created
Alternative Scenarios:	Working directory not a Git repo/ Issue repo already exists <ol style="list-style-type: none">1. Same as 12. Dev receives appropriate error message

Title (goal):	Create Issue
Actor(s):	Developer
Description (story):	Creates an issue inside the issue repository. Issue can now be used by other features e.g. print details.
Pre-Conditions:	Issue repository exists inside working directory.
Post-Conditions:	Issue exists inside issue repository.
Main Success Scenario:	<ol style="list-style-type: none">1. Dev types command to create issue providing the required cli arguments2. Success message notifies dev issue was created
Alternative Scenarios:	Working directory doesn't contain an issue repo <ol style="list-style-type: none">1. Same as 12. Error message informing dev that an issue repository hasn't been initialised Dev provides invalid cli arguments <ol style="list-style-type: none">1. Command for creating issue entered with invalid arguments2. Suitable error message output to developer

Title (goal):	Pull Issues
Actor(s):	Developer
Description (story):	Developer wants to receive issues and/or updates from a remote repository.
Pre-Conditions:	Issue repository exists inside working directory. Remote is a valid issue repository. Remote contains issues to receive.
Post-Conditions:	Developers repository is updated with issues from the remote. No data is incorrectly lost or modified.
Main Success Scenario:	<ol style="list-style-type: none"> 1. Dev types command to pull issues specifying the remote 2. Issues are pulled into the user's local repository
Alternative Scenarios:	<p>Remote doesn't contain any updates</p> <ol style="list-style-type: none"> 1. Same as 1 2. Developer receives message saying no updates exist on remote 3. Nothing is updated <p>Pulling from remote will create conflicts with user's local issue repository</p> <ol style="list-style-type: none"> 1. Same as 1 2. Issue receives message that will receive conflicts 3. User picks option A or B <ol style="list-style-type: none"> a. Dev chooses to resolve merge <ol style="list-style-type: none"> i. User resolves conflicts in issues ii. Commits changes to the repo b. Dev chooses to abandon merge <ol style="list-style-type: none"> i. User types command to abandon merge ii. Repo is reset to pre-pull state <p>Working directory doesn't contain an issue repo</p> <ol style="list-style-type: none"> 1. Same as 1 2. Error message informing dev that an issue repository hasn't been initialised

Title (goal):	Push Issues
Actor(s):	Developer
Description (story):	Developer wants to share issues from their repository to a remote repository.
Pre-Conditions:	Issue repository exists inside working directory. Remote is a valid issue repository. Local issue repo contains issues to push
Post-Conditions:	Remote repository is updated with issues if possible. Local repository is unchanged.
Main Success Scenario:	<ol style="list-style-type: none"> 1. Dev types command to push issues specifying the remote 2. Remote successfully is updated to contain issues from local repo 3. Dev receives success message saying remote was updated
Alternative Scenarios:	Remote is ahead of local repository/contains data local repo doesn't have <ol style="list-style-type: none"> 1. Same as 1 2. Remote rejects changes 3. Dev receives error message saying that the push update is not fast-forward and that they should fetch and merge the remote before re-pushing Working directory doesn't contain an issue repo <ol style="list-style-type: none"> 1. Same as 1 2. Error message informing dev that an issue repository hasn't been initialised

Title (goal):	Delete Issue
Actor(s):	Developer
Description (story):	Deletes an issue from the issue repository. The issue will not be visible to users unless the developer reverts to an old version (commit).
Pre-Conditions:	Issue repository exists inside working directory. An issue exists to delete.
Post-Conditions:	Issue has been deleted from issue repository.
Main Success Scenario:	<ol style="list-style-type: none"> 1. Dev types command to delete issue specifying the issue to delete 2. Success message notifies dev that the issue was removed from the repository
Alternative Scenarios:	Working directory doesn't contain an issue repo <ol style="list-style-type: none"> 1. Same as 1 2. Error message informing dev that an issue repository hasn't been initialised Dev provides non-existing issue to delete <ol style="list-style-type: none"> 1. Command for deleting issue entered with non-existing issue identifier specified 2. Suitable error message output to developer

Title (goal):	Update Issue Title
Actor(s):	Developer
Description (story):	Update the title of an existing issue to a new value. The issue will now display the new title and update its edited attribute.
Pre-Conditions:	Issue repository exists inside working directory. An issue exists to update.
Post-Conditions:	Title of issue has been updated.
Main Success Scenario:	<ol style="list-style-type: none"> 1. Dev types the command to update the title of an issue, specifying which issue to update and the new title 2. Success message notifies dev that the issue was updated
Alternative Scenarios:	<p>Working directory doesn't contain an issue repo</p> <ol style="list-style-type: none"> 1. Same as 1 2. Error message informing dev that an issue repository hasn't been initialised <p>Dev provides non-existing issue to update</p> <ol style="list-style-type: none"> 1. Command for creating issue entered with non-existing issue identifier specified 2. Suitable error message output to developer

Title (goal):	Update Issue Description
Actor(s):	Developer
Description (story):	Update the description of an existing issue to a new value. The issue will now display the new description and update its edited attribute.
Pre-Conditions:	Issue repository exists inside working directory. An issue exists to update.
Post-Conditions:	Description of issue has been updated.
Main Success Scenario:	<ol style="list-style-type: none"> 1. Dev types the command to update the description of an issue, specifying which issue to update and the new description 2. Success message notifies dev that the issue was updated
Alternative Scenarios:	<p>Working directory doesn't contain an issue repo</p> <ol style="list-style-type: none"> 1. Same as 1 2. Error message informing dev that an issue repository hasn't been initialised <p>Dev provides non-existing issue to update</p> <ol style="list-style-type: none"> 1. Command for creating issue entered with non-existing issue identifier specified 2. Suitable error message output to developer

Title (goal):	Update Issue Status
Actor(s):	Developer
Description (story):	Update the status of an existing issue to a new state. The issue will now show its new status and update its edited attribute.
Pre-Conditions:	Issue repository exists inside working directory. An issue exists to update.
Post-Conditions:	Status of issue has been updated.
Main Success Scenario:	<ol style="list-style-type: none"> 1. Dev types the command to update the status of an issue, specifying which issue to update 2. Success message notifies dev that the issue was updated
Alternative Scenarios:	<p>Working directory doesn't contain an issue repo</p> <ol style="list-style-type: none"> 1. Same as 1 2. Error message informing dev that an issue repository hasn't been initialised <p>Dev provides non-existing issue to update</p> <ol style="list-style-type: none"> 1. Command for creating issue entered with non-existing issue identifier specified 2. Suitable error message output to developer

Title (goal):	Add comment
Actor(s):	Developer
Description (story):	A dev can add a comment to an issue. The issue will now show this comment if the issue is viewed. The issue will update its edited status.
Pre-Conditions:	Issue repository exists inside working directory. An issue exists to add a comment to.
Post-Conditions:	Comment now attached to the issue.
Main Success Scenario:	<ol style="list-style-type: none"> 1. Dev types the command to add a comment to an issue, specifying which issue to add it to 2. Success message displayed to the informing them comment was added to the issue
Alternative Scenarios:	<p>Working directory doesn't contain an issue repo</p> <ol style="list-style-type: none"> 1. Same as 1 2. Error message informing dev that an issue repository hasn't been initialised <p>Dev provides non-existing issue to add comment to</p> <ol style="list-style-type: none"> 1. Command for creating comment entered with non-existing issue identifier specified 2. Suitable error message output to developer

Title (goal):	Add assignee
Actor(s):	Developer
Description (story):	A user can add an assignee to an issue. The issue will now show this user as an assignee if the issue is viewed. The issue will update its edited status. User is now responsible for dealing with the issue.
Pre-Conditions:	Issue repository exists inside working directory. An issue exists to add an assignee to.
Post-Conditions:	Assignee now assigned to the issue.
Main Success Scenario:	<ol style="list-style-type: none"> 1. Dev types the command to add an assignee to an issue, specifying which issue to add it to 2. Success message displayed to the dev informing them an assignee was added to the issue
Alternative Scenarios:	<p>Working directory doesn't contain an issue repo</p> <ol style="list-style-type: none"> 1. Same as 1 2. Error message informing dev that an issue repository hasn't been initialised <p>Dev provides non-existing issue to add assignee to</p> <ol style="list-style-type: none"> 1. Command for adding assignee entered with non-existing issue identifier specified 2. Suitable error message output to developer

Title (goal):	Remove assignee
Actor(s):	Developer
Description (story):	A user can remove an assignee from an issue. The issue will no longer show this user as an assignee if the issue is viewed. The issue will also update its edited status. User is now no longer responsible for the issue.
Pre-Conditions:	Issue repository exists inside working directory. An issue exists to remove an assignee from. A user is assigned to the issue.
Post-Conditions:	User now unassigned from the issue.
Main Success Scenario:	<ol style="list-style-type: none"> 1. Dev types the command to remove an assignee from an issue, specifying which issue to remove it from 2. Success message displayed to the dev informing them an assignee was remove from the issue
Alternative Scenarios:	<p>Working directory doesn't contain an issue repo</p> <ol style="list-style-type: none"> 1. Same as 1 2. Error message informing dev that an issue repository hasn't been initialised <p>Dev provides non-existing issue to remove assignee from</p> <ol style="list-style-type: none"> 1. Command for removing assignee entered with non-existing issue identifier specified 2. Suitable error message output to developer

Title (goal):	Add watcher
Actor(s):	Developer
Description (story):	A user can add a watcher to an issue. The issue will now show this user as a watcher if the issue is viewed. The issue will update its edited status. User will now receive notifications about the issue.
Pre-Conditions:	Issue repository exists inside working directory. An issue exists to add a watcher to.
Post-Conditions:	Watcher now added to the issue.
Main Success Scenario:	<ol style="list-style-type: none"> 1. Dev types the command to add a watcher to an issue, specifying which issue to add it to 2. Success message displayed to the dev informing them a watcher was added to the issue
Alternative Scenarios:	<p>Working directory doesn't contain an issue repo</p> <ol style="list-style-type: none"> 1. Same as 1 2. Error message informing dev that an issue repository hasn't been initialised <p>Dev provides non-existing issue to add watcher to</p> <ol style="list-style-type: none"> 1. Command for adding watcher entered with non-existing issue identifier specified 2. Suitable error message output to developer

Title (goal):	Remove watcher
Actor(s):	Developer
Description (story):	A user can remove a watcher from an issue. The issue will no longer show this user as a watcher if the issue is viewed. The issue will also update its edited status. User will now no longer receive notifications for the issue.
Pre-Conditions:	Issue repository exists inside working directory. An issue exists to remove a watcher from. A user is added as a watcher to the issue.
Post-Conditions:	User now unassigned from the issue.
Main Success Scenario:	<ol style="list-style-type: none"> 1. Dev types the command to remove a watcher from an issue, specifying which issue to remove it from 1. Success message displayed to the dev informing them a watcher was removed from the issue
Alternative Scenarios:	<p>Working directory doesn't contain an issue repo</p> <ol style="list-style-type: none"> 1. Same as 1 2. Error message informing dev that an issue repository hasn't been initialised <p>Dev provides non-existing issue to remove watcher from</p> <ol style="list-style-type: none"> 1. Command for removing watcher entered with non-existing issue identifier specified 2. Suitable error message output to developer

Title (goal):	Add attachment
Actor(s):	Developer
Description (story):	A user can add an attachment to an issue. The issue will now show this media as an attachment if the issue is viewed. The issue will update its edited status.
Pre-Conditions:	Issue repository exists inside working directory. An issue exists to add an attachment to.
Post-Conditions:	Attachment now contained in the issue.
Main Success Scenario:	<ol style="list-style-type: none"> 1. Dev types the command to add an attachment to an issue, specifying which issue to add it to 2. Success message displayed to the dev informing them an attachment was added to the issue
Alternative Scenarios:	<p>Working directory doesn't contain an issue repo</p> <ol style="list-style-type: none"> 1. Same as 1 2. Error message informing dev that an issue repository hasn't been initialised <p>Dev provides non-existing issue to add attachment to</p> <ol style="list-style-type: none"> 1. Command for adding attachment entered with non-existing issue identifier specified 2. Suitable error message output to developer

Title (goal):	Remove attachment
Actor(s):	Developer
Description (story):	A user can remove an attachment from an issue. The issue will no longer show this media as an attachment if the issue is viewed. The issue will also update its edited status.
Pre-Conditions:	Issue repository exists inside working directory. An issue exists to remove an attachment from. A piece of media is attached to the issue.
Post-Conditions:	Attachment is removed from issue
Main Success Scenario:	<ol style="list-style-type: none"> 1. Dev types the command to remove an attachment from an issue, specifying which issue to remove it from 2. Success message displayed to the dev informing them an attachment was removed from the issue
Alternative Scenarios:	<p>Working directory doesn't contain an issue repo</p> <ol style="list-style-type: none"> 1. Same as 1 2. Error message informing dev that an issue repository hasn't been initialised <p>Dev provides non-existing issue to remove attachment from</p> <ol style="list-style-type: none"> 1. Command for removing attachment entered with non-existing issue identifier specified 2. Suitable error message output to developer

Title (goal):	Show Issue
Actor(s):	Developer
Description (story):	A develop wants to view the full details of an issue so they use the show command.
Pre-Conditions:	Issue repository exists inside working directory. An issue exists to show details for.
Post-Conditions:	Full details of an issue output in terminal window
Main Success Scenario:	<ol style="list-style-type: none"> 1. Dev types command to show issue details, specifying the issue's id 2. Full details are displayed in terminal window
Alternative Scenarios:	<p>Working directory doesn't contain an issue repo</p> <ol style="list-style-type: none"> 1. Same as 1 2. Error message informing dev that an issue repository hasn't been initialised <p>Issue doesn't exist</p> <ol style="list-style-type: none"> 1. Dev types show command with non-existing issue's id 2. Error message displayed saying no issue with given ID

Title (goal):	Add tag
Actor(s):	Developer
Description (story):	Develop wants to add a descriptive tag to an issue e.g. label it as a bug or new feature
Pre-Conditions:	Issue repository exists inside working directory. An issue exists to add a tag to.
Post-Conditions:	Issue now has a tag attached to it.
Main Success Scenario:	<ol style="list-style-type: none"> 1. Dev types the command to add a tag to an issue, specifying which issue to add it to 2. Success message displayed to the dev informing them a tag was added to the issue
Alternative Scenarios:	<p>Working directory doesn't contain an issue repo</p> <ol style="list-style-type: none"> 1. Same as 1 2. Error message informing dev that an issue repository hasn't been initialised <p>Dev provides non-existing issue to add tag to</p> <ol style="list-style-type: none"> 1. Command for adding tag entered with non-existing issue identifier specified 2. Suitable error message output to developer

Title (goal):	Remove tag
Actor(s):	Developer
Description (story):	A user can remove a tag from an issue. The issue will no longer show this tag if the issue is viewed. The issue will also update its edited status.
Pre-Conditions:	Issue repository exists inside working directory. An issue exists to remove a tag from. A tag is added to the issue.
Post-Conditions:	Issue no longer has specified tag.
Main Success Scenario:	<ol style="list-style-type: none"> 1. Dev types the command to remove a tag from an issue, specifying which issue to remove it from 2. Success message displayed to the dev informing them a tag was removed from the issue
Alternative Scenarios:	<p>Working directory doesn't contain an issue repo</p> <ol style="list-style-type: none"> 1. Same as 1 2. Error message informing dev that an issue repository hasn't been initialised <p>Dev provides non-existing issue to remove tag from</p> <ol style="list-style-type: none"> 1. Command for removing tag entered with non-existing issue identifier specified 2. Suitable error message output to developer

Title (goal):	List issues
Actor(s):	Developer
Description (story):	User wants to output a list of issues contained in the repository. Can also filter which ones they see.
Pre-Conditions:	Issue repository exists inside working directory. At least 1 issue exists to be listed
Post-Conditions:	List of issues in issue repo displayed
Main Success Scenario:	<ol style="list-style-type: none"> 1. Dev types the command to list issues inside repo 2. Issue ID and title is displayed for issues inside repository
Alternative Scenarios:	<p>Dev wants to display more details than just id and title</p> <ol style="list-style-type: none"> 1. Dev types command to list issues with verbose flag 2. All information for every issue is output except for comments <p>Dev wants to filter issues by a certain attribute</p> <ol style="list-style-type: none"> 1. Dev types command to list issue with the attribute to sort with 2. Only issues matching the query are displayed <p>Dev wants to export data contained in issues to a CSV file for use in another program</p> <ol style="list-style-type: none"> 1. Dev types command to list issue and adds export to csv flag 2. Issues are listed into terminal and message output displaying where CSV file was created <p>Working directory doesn't contain an issue repo</p> <ol style="list-style-type: none"> 1. Same as 1 2. Error message informing dev that an issue repository hasn't been initialised

Title (goal):	Show help
Actor(s):	Developer
Description (story):	User wants to display help text for information on how to use a command of the system
Pre-Conditions:	None
Post-Conditions:	Help text for specified command is displayed in terminal
Main Success Scenario:	<ol style="list-style-type: none"> 1. Dev specifies command they want help text for 2. Help text is displayed in console
Alternative Scenarios:	<ol style="list-style-type: none"> 1. Dev specifies invalid command that they want help for 2. Error message telling dev that they entered invalid command

Appendix C – Detailed Test Strategy and Test Cases

C.1 Black Box Tests

As the amount of black box tests is extensive please go to file “blackBoxTests” to view them.

C.2 White Box Tests

Multiple unit tests were created for every method of the system. This was to ensure that every possible execution path was covered. In the figure below the line coverage and mutation coverage for the unit tests can be seen.

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage	Mutation Coverage
33	99% <div><div>1286/1297</div></div>	85% <div><div>506/592</div></div>

Breakdown by Package

Name	Number of Classes	Line Coverage	Mutation Coverage
cli	1	100% <div><div>166/166</div></div>	96% <div><div>25/26</div></div>
commands	18	99% <div><div>519/523</div></div>	84% <div><div>217/258</div></div>
csv	2	100% <div><div>32/32</div></div>	100% <div><div>12/12</div></div>
git	6	98% <div><div>356/363</div></div>	74% <div><div>128/172</div></div>
issueData	4	100% <div><div>151/151</div></div>	100% <div><div>69/69</div></div>
queries	2	100% <div><div>62/62</div></div>	100% <div><div>55/55</div></div>

The 15% of mutation coverage that is missing is due to code like the following examples:

```
24     ObjectId insert(CommitBuilder commit) throws IOException {
25         ObjectInserter objectInserter = repository.newObjectInserter();
26         ObjectId result = objectInserter.insert(commit);
27 1         objectInserter.flush();
28 1         return result;
29     }
```

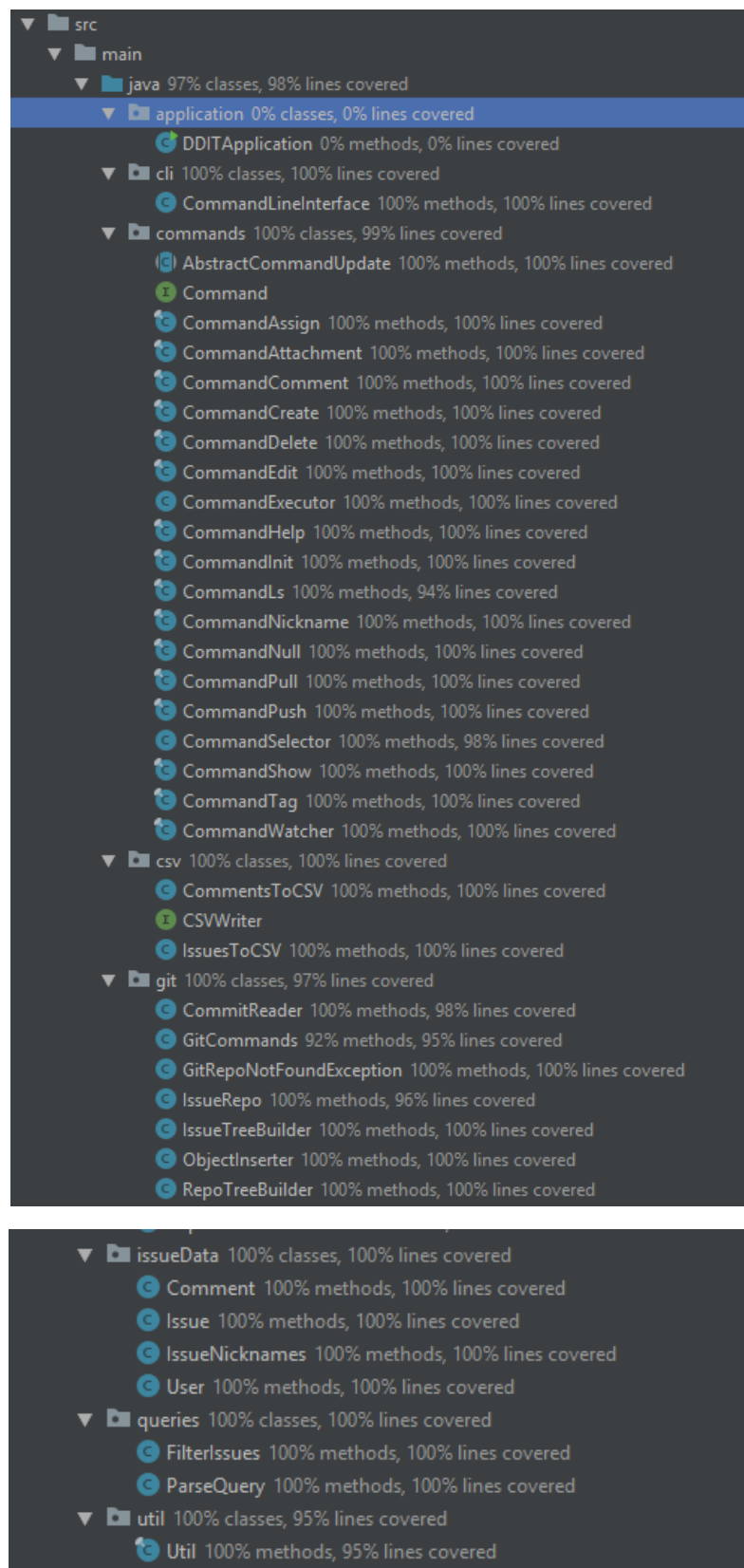
In the above snippet of code, mutating the code by removing the flush line causes no change to the result of the tests so will not cause the mutant to be killed. This is because calling flush only makes the created Git object visible inside the repository. It has no effect on the project’s code as it has a direct reference in the result variable. The JGit API docs recommend to always include this line.

```
79 1         try (RevWalk walk = new RevWalk(repo)) {
80             commit = walk.parseCommit(newCommit);
81 1             walk.dispose();
82 1         }
```

Another example of mutants not killed can be seen above. This try with resources block walks the commit graph of the repository. The try with resources block calls the close() method of the RevWalk after the code inside has finished executing. The only function calling close does is release the resources used by the RevWalk. The system does not rely on this method being called therefore there is no test for it. For this reason, the mutant is not killed.

The full mutation report can be found in the “mutation-report” directory of the submission files.

Below the coverage for each individual class in the system can be seen. The DDITApplication has 0% coverage as its simply a driver class instantiating and object and calling execute().



Appendix D—System Class Diagram

A high-quality image of the class diagram can be found in the submission files named “classdiagram”.

Several of the command classes have been excluded from the image, this is because it makes the class diagram completely unreadable. However, all these command classes interact the same as the few command classes left in the diagram.

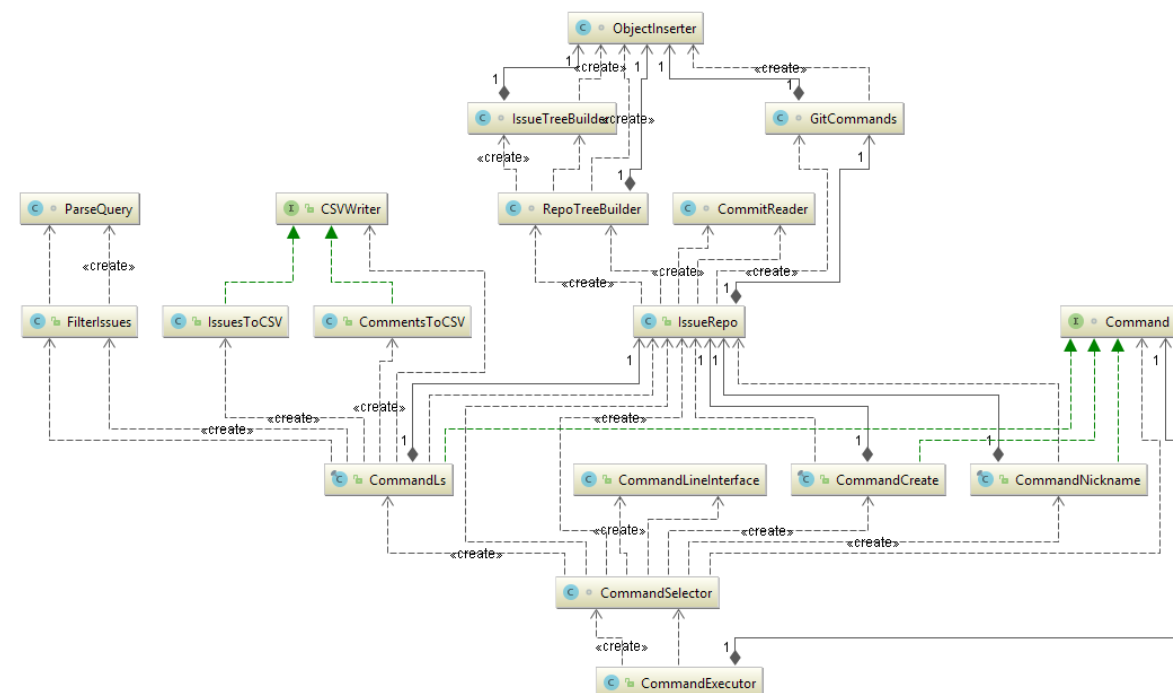
The POJO objects Issue, Comment and User have been omitted also as a majority of the classes need to know about these objects. This is to make the diagram clearer.

The system can be split into decoupled parts:

There is the CommandSelector and CommandExecutor classes which are responsible for handling the user input and executing the correct command.

The command classes interact with the IssueRepo class which is the public interface for commands to use git functionality. The CommandLs class also interacts with the query language and the issue exporter class.

The IssueRepo is the public interface to use Git functionality. Only this class interacts with the internal Git functionality.



Powered by yFiles

Appendix E – User Guide and Quick Start Guide

E.1 Quick Start Guide

The purpose of this quick start guide is to make the user familiar with the basic commands of the ITS as fast as possible. This guide assumes that the installation guide has been followed and that the user's current working directory is inside of a Git directory.

E.1.1 Initialise the Issue Repository

An issue repository must be created before any of the commands can do anything. This is done by entering the command:

```
git ddit init
```

E.1.2 Create an issue

To create an issue, enter the command shown below - changing what is in the quotation marks to be your title and description (remember arguments with spaces must be in quotation marks):

```
git ddit new --title "this is my title" --desc "this is my description"
```

E.1.3 Pushing Issues to a Remote Repository

To push issues to a remote repository, the remote must be specified. If the remote is not stored on the local machine, an SSH key must have been set up and the remotes SSH URL must be specified.

```
git ddit push -r /remote/repo
```

E.1.4 Pulling Issues from a Remote Repository

To pull issues from a remote repository, the remote must be specified. If the remote is not stored on the local machine, an SSH key must have been set up and the remotes SSH URL must be specified.

```
git ddit pull -r /remote/repo
```

E.1.5 Adding a Nickname

Since the identifier of an issue is a SHA256 hash value by default, it makes it easier to add a customised nickname to the issue to refer to the issue. To do this enter the following command:

```
git ddit nickname --add <hash> nickNameToUse
```

E.1.6 Commenting on an Issue

To comment on an issue, use the following command. The nickname created in the previous instruction is used. The original hash or nickname can be specified in the "-id" flag.

```
git ddit comment --id myIssue -m "this is my comment"
```

E.1.7 Listing issues

The simplest way to list issues in the repository is using the following command.

```
git ddit ls
```

E.1.8 Showing an Issues Full Data

To view all information in an issue, use the following command.

```
git ddit show --id myIssue
```

E.2 User Guide

The purpose of this user guide is to give a comprehensive tutorial on how to use the system. This includes using every command and every option they have to offer.

This guide assumes that the installation guide has been followed and that the user's current working directory is inside of a Git directory.

E.2.1 Command Usage

Every command of the ITS starts with "git ddit" assuming that the installation guide has been followed properly. The full list of commands is shown in the next section of this guide. To use a command the basic format: "git ddit <commandName> <options>" is used. An example is "git ddit show --id myissue".

Any argument that contains white space must be surrounded with quotation marks. E.g. to add a title with multiple words, enter: --title "this is an example". Instead of: --title this is an example

E.2.2 Help

The most basic option for all commands is to use "-h" or "--help". This will provide a help text for the command displaying all options available. If no command is supplied, then the system will show the main menu help which is an overview of the commands the system has to offer.

E.2.3 Initialising an Issue Repository

Before doing anything in the system, the issue repository must be initialised. This is done with the "init" command. Example shown below. This command will create a new branch named "issues" in the repository. If an "issues" branch already exists, the system will not recreate the branch. The system will let the user know that it cannot initialise.

```
git ddit init
```

E.2.4 Creating a New Issue

New is the command used to create an issue. It has non-optional options "title" and "desc" (description). The reason these must be specified is because issues must be created with a title and description.

```
git ddit new --title "this is my title" --desc "this is my description"
```

E.2.5 Deleting an Issue

To delete an issue the command is simple. It has one required option and nothing else. This is the ID of the issue to delete. The ID can be the issue's hash value or nickname (nicknames defined later in this guide).

```
git ddit delete --id <hash>
```

E.2.6 Commenting on an Issue

To comment on an issue the required options are the ID of the issue to comment on and the message. An example of adding a comment to an issue is the following:

```
git ddit comment --id hash --message "this is my comment"
```

E.2.7 Listing all Issues

Using the ls command with no arguments by default will list all issues in the repository. The format of the ls command is the follow:

```
git ddit ls
```

E.2.8 Showing an Issues Full Details

To display the full details of an issue is simple. Only one option is required, and this is the ID of the issue to show:

```
git ddit show -id <hash>
```

E.2.9 Pushing Issues to a Remote Repository

To push issues to a remote repository, the remote must be specified. If the remote is not stored on the local machine, an SSH key must have been set up and the remotes SSH URL must be specified.

```
git ddit push -r /remote/repo
```

E.2.10 Pulling Issues from a Remote Repository

To pull issues from a remote repository, the remote must be specified. If the remote is not stored on the local machine, an SSH key must have been set up and the remotes SSH URL must be specified.

```
git ddit pull -r /remote/repo
```

E.2.11 Updating an Issues Status

To close or reopen an issue, the edit command is used. The ID of the issue must be specified along with whether to open or close the issue.

```
git ddit edit -id <hash> --status close
```

E.2.12 Updating an Issues Title

To update an issue's title, the edit command is used. The id of the issue must be specified along with the new title. This will update the title of the issue to be "new title"

```
git ddit edit -id <hash> --title "new title"
```

E.2.13 Updating an Issues Description

To update an issue's description, the edit command is used. The id of the issue must be specified along with the new description. This will update the description of the issue to be "new description":

```
git ddit edit -id <hash> --desc "new description"
```

E.2.14 Listing Issues with a Filter

To list issues with a filter the ls command is used the query option. There are many possible query combinations so for the full help refer to the help text in H.8. This will show issues with title: hello, description: world and a status of open:

```
git ddit ls -q title:hello desc:world status:open
```

E.2.15 Exporting CSV files

To export a CSV file a special option is used with the ls command. The issues contained in the csv can be filtered using the query language. -csv will create a file containing issue data and -csvc will create a file containing comment data.

```
git ddit ls -csv <path> -csvc <path>
```


E.2.16 Adding a Tag to an Issue

To add a tag to an issue, the tag command is used. Multiple tags can be added at once if they are separated with a comma. This will add the tags "bug" and "feature" to the issue:

```
git ddit tag -id <hash> --add "bug, feature"
```

E.2.17 Removing a Tag from an Issue

To remove a tag from an issue, the tag command is used. Multiple tags can be removed at once if they are separated with a comma. This will remove the tag "bug" from the issue:

```
git ddit tag -id <hash> -rm "bug"
```

E.2.18 Adding an Attachment to an Issue

To add a media attachment to an issue, the attach command is used. Attachments can be a max of 20Mb. This will add the jpg file "diagram" to the issue:

```
git ddit attach -id <hash> --add /home/diagram.jpg
```

E.2.19 Removing an Attachment from an Issue

To add a media attachment to an issue, the attach command is used. This will remove the file with name diagram.jpg from the issue:

```
git ddit attach -id <hash> -rm diagram.jpg
```

E.2.20 Adding an Assignee to an Issue

To add an assignee to resolve issues use the assign command.

```
git ddit assign -id <hash> --add --name "jack" --email "my@email.com"
```

This will assign the user with name jack and email my@email.com to the issue.

E.2.21 Removing an Assignee from an Issue

To remove an assignee from an issue, use the assign command.

```
git ddit assign -id <hash> -rm --name "jack" --email "my@email.com"
```

This will un-assign the user with name jack and email my@email.com to the issue.

E.2.22 Listing all Assignees on an Issue

To list all the assignees on an issue, use the assign command.

```
git ddit assign -id <hash> -rm --name "jack" --email "my@email.com"
```

E.2.23 Adding a Watcher to an Issue

To add a watcher to an issue, use the watch command.

```
git ddit watch -id <hash> --add --name "jack" --email "my@email.com"
```

This will add the user with name jack and email my@email.com to the issue as a watcher.

E.2.24 Removing a Watcher from an Issue

To remove a watcher from an issue, use the watch command.

```
git ddit watch -id <hash> --rm --name "jack" --email "my@email.com"
```

This will remove the user with name jack and email my@email.com from the issue as a watcher.

E.2.25 Listing all Watchers on an Issue

To list all the watchers on an issue, use the watch command.

```
git ddit assign -id <hash> -rm --name "jack" --email "my@email.com"
```

E.2.26 Adding a Nickname to an Issue

Since the identifier of an issue is a SHA256 hash value by default, it makes it easier to add a customised nickname to the issue to refer to the issue. To do this enter the following command:

```
git ddit nickname --add <hash> nickNameToUse
```

E.2.27 Updating a Nickname on an Issue

To update a nickname, use the nickname command. The old nickname can be used or the hash to specify which nickname to update. This will update the nickname of the issue to "new nickname":

```
git ddit nickname --update oldNickname newNickname
```

E.2.28 Removing a Nickname from an Issue

To remove a nickname, use the nickname command. The nickname can be used or the hash to specify which nickname to remove. This will remove the nickname from the issue:

```
git ddit nickname --remove nickname
```

Appendix F – Installation Guide

If a JAR file is already sourced, then skip the first section.

F.1 Obtaining a JAR from Source Code

Source code available [here](#).

To install the ITS project from source code the following prerequisites must be installed:

- JDK version 10+
- Apache Maven version 3.5.2+

Once these are installed, navigate a shell's working directory to the directory the source code is stored. Now run the following command: "mvn clean compile assembly:single". The jar will then be created inside the target directory. This JAR can then be placed in any directory of user's choice.

F.2 Setting up JAR to use

To execute the JAR the Java Runtime Environment version 10+ must be installed (this can be installed separately but is also packaged with the JDK).

Other than the JRE there are no other prerequisites to use the JAR however, it is extremely recommended to have the users name and email specified in the ".gitconfig" file located in the user's home directory. Usually this will already be set up as the JAR is used from within Git repositories and to obtain a project's repository, users will have already set up Git.

Next create a shell script named "git-ddit" in a directory of user's choice or download a copy of the script [here](#).

Inside this script enter the following (changing the "path/to/jar" to the fully qualified file path of the JAR):

```
#!/bin/bash# Usage: git ddit
if [ -d .git ]; then
    java -jar /path/to/jar/ddit.jar "$@"
else
    git rev-parse --git-dir 2> /dev/null;
    echo "Not in a git directory";
fi;
```

Next this script must be given permissions to be an executable. This can be done with the command "chmod +x /path/to/script/git-ddit".

The last thing is to add the directory that contains this script to the users \$PATH variable. This can be done by adding: export PATH="directory/of/script:\$PATH" to the "~/.bashrc" or "~/.profile" file.

To check everything is set up correctly, navigate into a Git repository and type "git ddit". The main menu help text of the system should be printed in the terminal.

Appendix G – Maintenance Guide

G.1 Adding a Feature

As the software architecture uses a package by feature approach. When adding a new feature, create all code relevant to the new feature in a new package inside `src/main/java/`.

When unit testing this new feature create a new package inside `src/test/java` with the same name as the package created in the `src/main/java` directory.

Add the code that calls upon the new functionality inside the relevant command. If it is a new feature, then follow G.2.

G.2 Creating a Command

When creating a new command, create a new class inside the `src/main/java/commands` directory. Make sure the command implements the `Command` interface and then fill out functionality inside the `execute` method.

Inside the `src/main/java/CLI` package, update the class `CommandLineInterface`. In this class define the input options that should be available to the user.

Write a help text for the command and store it in `src/main/resources`. Update the `CommandHelp` class in `src/main/java/commands` and add the path to this file.

Update the switch statement inside `CommandSelector` to pass input args to the CLI option parser for the new command.

Create tests for new classes inside the mirrored file structure in `src/test/java/*`.

Appendix H – Command Help Texts

In this appendix the help text of every command is shown. To read in text file format go to the [source code](#) and navigate to directory “src/main/resouces/help”.

H.1 Main Menu

Can be displayed by entering “git ddit”, “git ddit -h” or “git ddit --help”

```
DDIT - A Distributed and Decentralised Issue Tracking System (ITS)
```

```
Usage:
```

```
git ddit <command> [<args>] [--help]
```

```
Commands are:
```

```
Initialising ITS
```

```
init      Create an issues branch in the repository
```

```
Creation/deletion of issues
```

```
delete    Delete an issue
```

```
new       Create a new issue
```

```
Examine issues
```

```
ls        List issues in the repository
```

```
show      Display the full details of a single issue
```

```
Collaborate
```

```
pull      Fetch and merge issues from a remote repository
```

```
push      Update a remote with associated issues
```

```
Update issues
```

```
assign    Assign an issue to a user
```

```
attach    Add an attachment to an issue e.g. an image
```

```
comment   Comment on an issue
```

```
tag       Add a tag to an issue e.g. "Bug"
```

```
update    Update the title, description or status of an issue
```

```
watcher   Add a watcher to an issue
```

H.2 Assign Help

Can be displayed by entering "git ddit assign -h" or "git ddit assign --help"

DDIT - git ddit assign

Important: Use quotation marks for args with whitespace

Usage:

```
git ddit assign [-id <hash> <--list | --add | --remove> [--name <name>
--email <email>]] [--help]
```

Description:

Assigns the issue to specified user. An assignee is someone who is working on the issue.

Options:

-a, --add	Tell system to add the user to an issue
-e, --email <email>	Email of the user being added
-h, --help	Show this help screen
-id <hash>	ID of issue to assign user to
-ls, --list	List all assignees for an issue
-n, --name <name>	Name of user being added
-rm, --remove	Tell system to remove the user from an issue

Example:

```
git ddit assign -id <hash> --name "jack" --email "my@email.com"
This will assign the user with name jack and email my@email.com to the
issue.
```

```
git ddit assign -id <hash> -rm --name "jack" --email "my@email.com"
This will un-assign the user with name jack and email my@email.com to
the issue.
```

H.3 Attach Help

Can be displayed by entering "git ddit attach -h" or "git ddit attach --help"

```
DDIT - git ddit attach
```

Usage:

```
git ddit attach [-id <hash> <--add <path-to-file> | --remove  
<attachment-name> >] [--help]
```

Description:

Adds a file to this issue as an attachment, this can be anything such as an image.

Options:

-a, --add <path-to-file>	Add a file to the issue
-h, --help	Show this help screen
-id <hash>	ID of issue to add attachment to
-rm, --remove <attachment-name>	Remove a file from issue

Example:

```
git ddit attach -id <hash> -a /home/diagram.jpg  
This will add the jpg file "diagram" to the issue
```

```
git ddit attach -id <hash> -rm diagram.jpg  
This will remove the file with name diagram.jpg from the issue
```

H.4 Comment Help

Can be displayed by entering "git ddit comment -h" or "git ddit comment --help"

DDIT - git ddit comment

Important: Use quotation marks for args with whitespace

Usage:

```
git ddit comment <[-id <hash> -m <message>] [--help]>
```

Description:

Creates a new comment on the issue with the given ID.

Options:

-h, --help	Show this help screen
-id <hash>	ID of issue to comment on
-m, --message <msg>	Message to leave in the comment

Example:

```
git ddit comment -id <hash> -m "this is a new comment"
```

This will create a new comment on issue with given hash and message:
"this is a new comment".

H.5 Delete Help

Can be displayed by entering "git ddit delete -h" or "git ddit delete --help"

DDIT - git ddit delete

Usage:

```
git ddit delete <[-id <hash>] [--help]>
```

Description:

Deletes an issue from the issue repository.

Options:

```
-id <arg>      ID of issue to delete  
-h, --help    Show this help screen
```

Example:

```
git ddit delete <hash>
```

This will delete the issue that has the unique hash

H.6 Edit Help

Can be displayed by entering "git ddit edit -h" or "git ddit edit --help"

DDIT - git ddit edit

Important: Use quotation marks for args with whitespace

Usage:

```
git ddit edit <[-id <hash> <--title <title> | --desc <desc> | --status  
<open|close> >] [--help]>
```

Description:

Updates the specified attribute of an issue with the new specified value.

Options:

-d, --desc <desc>	Update description of issue
-h, --help	Show this help screen
-id <hash>	ID of the issue to update
-s, --status <status>	Update status of the issue
-t, --title <title>	Update title of issue

Example:

```
git ddit edit -id <hash> -d "new description"
```

This will update the description of the issue to be "new description"

```
git ddit edit -id <hash> -t "new title"
```

This will update the title of the issue to be "new title"

```
git ddit edit -id <hash> -s close
```

This will update the status of the issue to be closed

H.7 Init Help

Can be displayed by entering "git ddit init -h" or "git ddit init --help"

```
DDIT - git ddit init
```

Usage:

```
git ddit init [-h | --help]
```

Description:

Creates an empty issues branch inside an existing Git repository.

Options:

-h, --help Show this help screen

Example:

```
git ddit init
```

H.8 Ls Help

Can be displayed by entering "git ddit ls -h" or "git ddit ls --help"

DDIT - git ddit ls

Important: Separate conditions with a space, values with white space and/or dates should be surrounded by quotation marks

Usage:

```
git ddit ls [-q <query>...] [--verbose] [-csv] [--help]
```

Description:

Lists issues in the issue repository. Can add a query to filter issues, displaying only issues that match the conditions.

Options:

-csv <path>	Creates a csv containing issue information
-csvc <path>, --csv-comments <path>	Creates a csv containing comment information
-q, --query <arg>...	Description for new issue
-v, --verbose	Show full details of issues

Conditions:

Format of condition is: qualifier:value

Qualifiers are:

title, description, status, tag, assignee, watcher, creator, comment, created, edited

Value format varies depending on qualifier:

title:"title contains"	This will display issues that contain the value string in the qualifier
desc:"description contains"	
tag:"tag"	

status:open | status:closed These are the two options for the status qualifier - displaying issues with open/closed status

watcher:name watcher:email	Displays issues with a watcher/creator/assignee with the specified name or email
creator:name creator:email	
assignee:name assignee:email	

For qualifiers "created" and "edited" use of <, >, <=, and >= can be used:

If using <, >, <=, or >= then must be surrounded by " "

created:"[< > >= <=]dd/MM/yyyy"	Displays issues created/edited on the date given if no equality operator given else will display accordingly
edited:"[< > >= <=]dd/MM/yyyy"	

Example:

```
git ddit ls -q title:hello desc:world status:open
```

This will show issues with title: hello, description: world and a status of open.

```
git ddit ls
```

This will show all issues

```
git ddit ls -q created:<03/02/2019 title:"hello world"
```

This will show issues created before 03/02/2019 with the string "hello world" in the title

H.9 New Help

Can be displayed by entering "git ddit new -h" or "git ddit new --help"

DDIT - git ddit new

Important: Use quotation marks for args with whitespace

Usage:

```
git ddit new <[-t <title> -d <desc>] [--help]>
```

Description:

Creates a new issue on the issue branch with given title and description with an open status.

Options:

-d, --desc <arg>	Description for new issue
-h, --help	Show this help screen
-t, --title <arg>	Title for new issue

Example:

```
git ddit new -t "hello" -d "world"
```

This will create a new issue with title: hello and description: world

H.10 Nickname Help

Can be displayed by entering "git ddit nickname -h" or "git ddit nickname --help"

DDIT - git ddit nickname

Important: Use quotation marks for args with whitespace

Usage:

```
git ddit nickname [<--add <hash> <nickname> | --update <hash |  
nickname> <new nickname> | --remove <hash|nickname> > | --list] [--help]>
```

Description:

Adds nicknames to issues. Nicknames make it easier to identify issues to users compared to remembering a hash.

Options:

-a, --add <hash> <nickname> an issue	Add a nickname to
-h, --help screen	Show this help
-ls, --list nicknames for every issue	List all the
-u, --update <hash nickname> <new nickname> an issue	Update nickname of
-rm, --remove <hash nickname> nickname of an issue	Remove the

Example:

```
git ddit nickname --add <hash> nickname  
This will add a nickname "nickname" to the issue
```

```
git ddit nickname --update <hash> "new nickname"  
This will update the nickname of the issue to "new nickname"
```

```
git ddit nickname --remove <hash>  
git ddit nickname --remove nickname  
This will remove the nickname from the issue
```

```
git ddit nickname --list  
This will list all the nicknames for issues in the repository
```

H.11 Pull Help

Can be displayed by entering "git ddit pull -h" or "git ddit pull --help"

```
DDIT - git ddit pull
```

Usage:

```
git ddit pull <[-r <remote>] [--help]>
```

Description:

Pulls issues on remote. If there is a merge conflict an editor will open for the user to resolve.

Options:

```
-r, --remote <remote>    Remote location - to pull from  
-h, --help                Show this help screen
```

Example:

```
git ddit pull /home/user/project/  
This will pull the issues that are stored on the remote
```

H.12 Push Help

Can be displayed by entering "git ddit push -h" or "git ddit push --help"

DDIT - git ddit push

Usage:

git ddit push <[-r <remote>] [--help]>

Description:

Pushes issues to a remote. Changes will be rejected if not fast-forward merge.

Options:

-r, --remote <remote> Remote location - to push to
-h, --help Show this help screen

Example:

git ddit push /home/user/project/
This will push the issues that are stored to the remote

H.13 Show Help

Can be displayed by entering "git ddit show -h" or "git ddit show --help"

DDIT - git ddit show

Usage:

```
git ddit show <[-id <hash>] [--help]>
```

Description:

Shows all stored data in an issue including comments

Options:

```
-id <arg>      ID of issue to show  
-h, --help     Show this help screen
```

Example:

```
git ddit show <hash>  
This will show all of the issues data
```

H.14 Tag Help

Can be displayed by entering "git ddit tag -h" or "git ddit tag --help"

DDIT - git ddit tag

Important: Use quotation marks for args with whitespace, multiple tags can be added and removed at once separate with comma

Usage:

```
git ddit tag <[-id <hash> <--add <tag,...> | --remove <tag,...> >] [--help]>
```

Description:

Adds or removes a tag to an issue. A tag is used to categorise the issue e.g. "bug", "feature" or "enhancement".

Options:

-a, --add <tag>	Add a tag to the issue
-h, --help	Show this help screen
-id <hash>	ID of the issue to add tag to
-rm, --remove <tag>	Remove the tag from the issue

Example:

```
git ddit tag -id <hash> -a "bug, feature"
```

This will add the tags "bug" and "feature" to the issue.

```
git ddit tag -id <hash> -rm "bug"
```

This will remove the tag "bug" from the issue.

H.15 Watch Help

Can be displayed by entering "git ddit watch -h" or "git ddit watch --help"

DDIT - git ddit watch

Important: Use quotation marks for args with whitespace

Usage:

```
git ddit watch [-id <hash> <--list | --add | --remove> [--name <name> -  
-email <email>]] [--help]
```

Description:

Adds the specified user as a watcher to the issue. A watcher is someone who will receive notifications (if setup) about any changes to the issue.

Options:

-a, --add	Tell system to add the user to an issue
-e, --email	Email of the user being added
-h, --help	Show this help screen
-id <hash>	ID of issue to add watcher to
-ls, --list	List all assignees for an issue
-n, --name	Name of user being added
-rm, --remove	Tell system to remove the user from an issue

Example:

```
git ddit watch -id <hash> -a --name "jack" --email "my@email.com"  
This will add the user with name jack and email my@email.com to the  
issue as a watcher.
```

```
git ddit watch -id <hash> -rm --name "jack" --email "my@email.com"  
This will remove the user with name jack and email my@email.com from  
the issue as a watcher.
```