UNIVERSITY OF Southampton
School of Electronics and Computer Science

Electronics and Computer Science
Faculty of Physical Sciences and Engineering
University of Southampton

Emily Langdon Shepherd

28th April 2015

Creating a Distributed Peer-Approval Code Collaboration Model

Project supervisor: Dr Kirk Martinez
Second examiner: Dr Su White

A project report submitted for the award of
Computer Science, MEng

Abstract

This paper examines the version control software currently available, and gives a detailed analysis of the technical considerations needed when developing an extension to Git, the most popular of these tools. The report discusses various code peer approval tools, and notes that these all rely on centralised servers or workflows. It introduces the concepts of quantifiable reputation and automated user privilege administration, techniques which are utilised within community sites such as Ebay, Stack Exchange and Wikipedia. With this research, the paper proposes a specification for a decentralised peer approval tool, termed Legit, and explains the design decisions that this involved. Finally, a review of the project management and testing is given, and the specification is evaluated by contrasting it to modern day corporate and open source coding environments. The paper finishes with some suggestions for future work and research, and concludes that, by structuring communication and standardising common tasks, a decentralised peer approval workflow is potentially able to drastically reduce the administrative overhead required in many software engineering projects and may encourage teams to bond more effectively.

Contents

Tables and Figures

Acknowledgements

I would to thank Dr Kirk Martinez for stepping in as my supervisor at short notice in the middle of the academic year, and Dr Tim Chown for his initial support and advice at the start of the project.

Chapter 1

Introduction

Software development is an important aspect of Computer Science, and as such there are many strategies, techniques and programs which aim to formalise and structure this process. This issue becomes more complex when development is carried out in large global teams which is "*rapidly becoming the norm for technology companies*" [1]. Members of such groups may have a wide variety of cultures, languages, skills and coding preferences and habits. In open source projects, for example, quite literally anyone, from any background, can contribute. The Team Software Process[1] (TSP) Body of Knowledge (BOK) aptly describes the key issue here: "*Communication is the single most important element in both building and maintaining teams*" [2].

This is a wide area of discussion, as teams may vary enormously in size, purpose, governance and structure, all of which may have drastic effects on their dynamic, and the way in which they organise a given project. Despite these differences, several key issues need to be addressed for most projects: how are project members picked and introduced to a team, are they all of equal standing within the group? How are tasks shared out among contributors? How are the aims and goals of a project decided, and how are code contributions validated against these?

The way in which these questions are answered defines how a software team functions, and may ultimately solidify their success, or indeed their failure. Some projects may choose to address these issues manually, by subscribing to a software development strategy such as Agile, and possibly assigning a person, or group of people, to monitor the rest of the team. Projects also often turn to

[1] The Team Software Process was developed in the late 1990s, and is a framework of strategies intended to aid both developers and managers in the software development process. It is comparable to Agile, and further reading on this topic is given in the Bibliography.

software to aid in this process, of which there is a wide array available: some open source and free, others proprietary and commercial.

Ultimately, this normally involves some form of executive model, in which a person, or group of people, have permissions to commit changes to a project. There may be an organisational hierarchy within this group, either defined by management status or enforced by third party software. Users outside this group are normally not allowed to make edits directly; they are either excluded from the project altogether, or they may work amongst themselves to reach an agreement, which a member of the executive team will then accept into the main project.

1.1 Project Goals

This report first analyses the tools which currently exist to aid developers to organise their work, predominantly focusing on Git, a popular tool for managing software version control, and the tools based upon it. This includes how users are monitored within a project, and the processes by which code is vetted.

Secondly, team development will be looked at in a wider context, by investigating the way in which other group based organisations, which may not be directly involved in software development, employ peer review systems to collaboratively determine trust and to achieve their goals without a reliance on an executive model, or at least to a lesser degree. This will explore the benefits of user voting, and the way in which these systems detect and mitigate against abuse, and introduces the concept of quantifiable "reputation" which an automated system may use as a basis on which to grant user privileges.

This paper is submitted with an accompanying software package which attempts to implement these principles within the context of code development (See Appendix C) specifically on top of a project managed by Git. The effectiveness of this in a decentralised system is evaluated by contrasting its use to the centralised and decentralised workflows which users may abide by when using Git on its own, and the centralised workflows enforced by current peer approval software.

Chapter 2

Git Version Control

Git is a piece of software designed to manage code development by keeping track of all changes, updates and improvements, with an annotated history to serve as an audit trail[2]. It comes under the class of software known as *version control* [3].

2.1 Overview of Git

Git takes many paradigms (commits, branching and merging) from existing revision control software projects with a key focus of speed and scalability; an important aim being to protect developers from facing cumbersome and lengthy procedural overhead when working with code. Tests performed by the team of developers at Mozilla showed Git to be able to identify and display changes between commits at least 5 times quicker, and often reaching up to 100 times quicker, than comparable products [4]. This significant speed directly improves Git's scalability over other products, as it can process many more source files and commits before it becomes unacceptably slow. In its stated purpose and specification, therefore, Git performs well.

2.2 History and Popularity of Git

Git was developed by Linus Torvalds in 2005 after the Linux Project lost its license to use BitBucket, a commercial revision control platform [5] [6] [7]. Torvalds (together with a group of other Linux contributors), created Git with the purpose of creating a fast version control system, which could be used for free for the Linux Project. As of 2014, the Eclipse Foundation reported that 43% of developers in their survey responded that they use Git or a Git-based product[3] as their source

[2] The homepage for the Git Project: https://git-scm.herokuapp.com/

[3] Git-based products, GitHub and Gitlab, are tools built on top of Git, either adding extra functionality or a visual user interface.

control software of choice [8]. In that year, Git was the most used product, overtaking the previous developer favourite: Subversion.

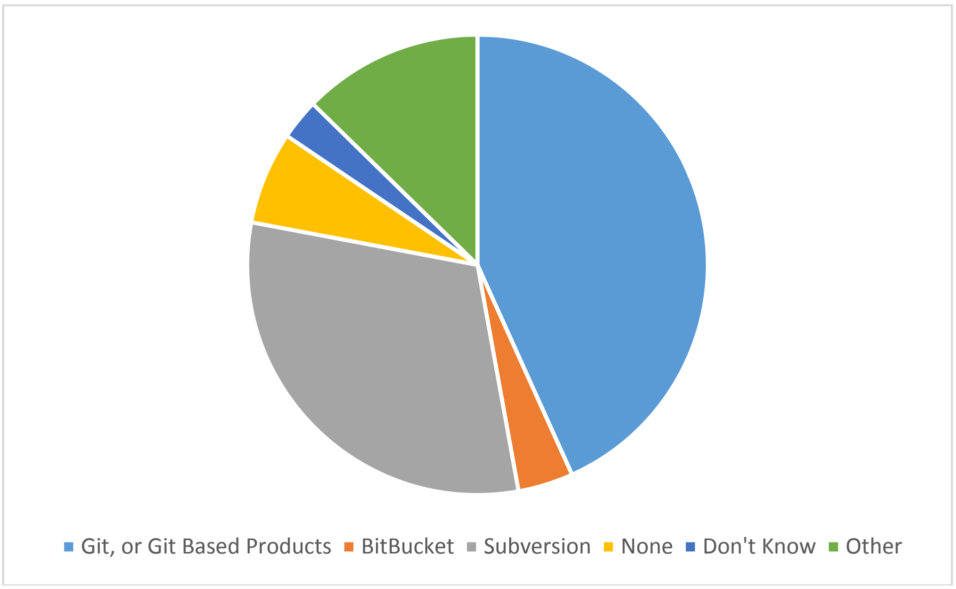

Figure 1: Primary source code management system for developers in 2014[4]

2.3 Terms from Git

A Git project is contained within a *repository*, the total store of all changes, point-in-time snapshots[5] and meta-data, for the duration of the entire project. Each developer who wishes to work on a project will have a local copy of the project's repository for their own development.

Within Git, and many other version control software packages, each atomic change is known as a *commit*, which is a snapshot of the entire code at a point in time. Multiple files can be changed in a single commit, and there is some meta-data associated with it, such as the time of the commit, the name of the author and a written explanation of the changes made. Each commit typically contains a reference to the previous one, termed its *parent* commit; this creates a line of commits going back in time, and is known as a *branch*.

[4] "Other" includes: IBM and Microsoft Products, CVS, Perforce, Mercurial and several niche software packages

[5] For recent commits, a complete snapshot of changed files is stored. This is one of the ways Git achieves such high speeds. For older commits, changes are stored as a delta, to save disk space.

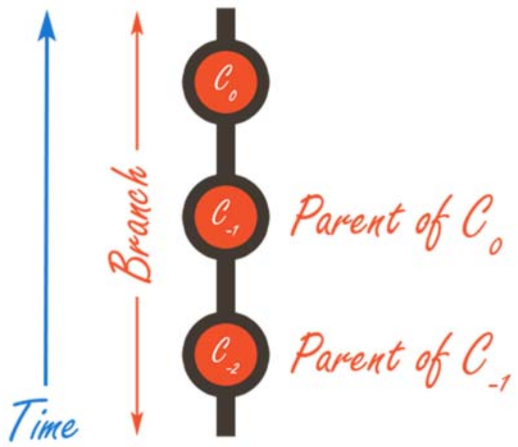

Figure 2: Three commits in a branch

Branches may stem from commits on other branches – where this occurs, the first commit of each of the subsidiary branches shares the same parent commit in the original branch. Branches may be merged together, to create a single branch going forward – in this case the first commit in the new branch will have multiple parent commits.

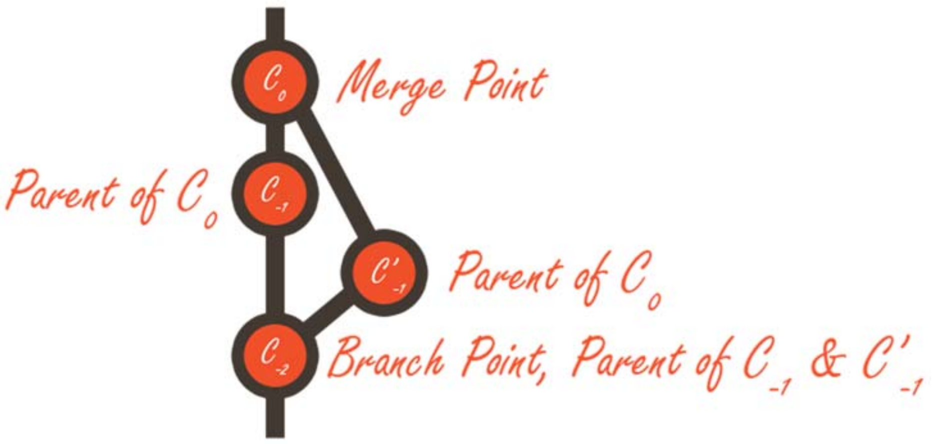

Figure 3: A commit branching from the main branch, then merging back in

Occasionally there is a need to move a branch, changing the point at which it stems from another. This is known as a *rebase* operation and is a complex task as it requires changing the parent of the first commit and updating each of the commits in the rebased branch. Clearly, in large projects with multiple branches, the network has the potential to become quite complex, especially as branches can overlap one another.

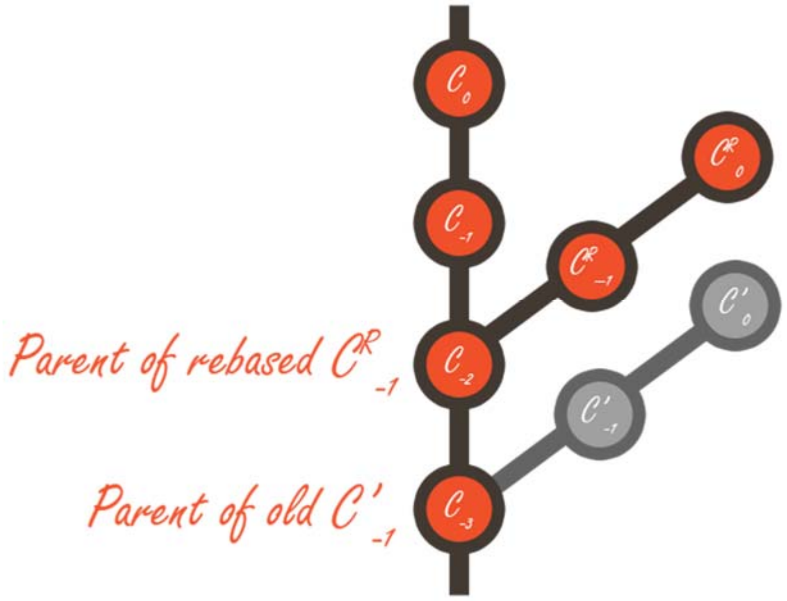

Figure 4: Branch being rebased by one commit

Git allows files, directories and commits to be annotated with extra meta-data, known as *tags*. They are typically used to identify important commits, such as bug fixes and new feature releases.

Developers working on a project often need to synchronise their repository with another computer, possibly that of a teammate or a central server. When the computer that is being updated initiates this, it is known as a "pull", whereas when it is the updating computer, it is termed a "push". When a computer receives a push from another, Git will automatically check the integrity of the contents; it allows the owner of the receiving computer to define their own validation behaviour program or script. This ability does not exist when performing a pull, the updated computer cannot perform any user-defined verification.

2.4 Git's Workflow

Git is a distributed system – the entire commit history exists on every computer that uses it. This differs from various other source control systems such as Subversion, which relies upon a centralised server. It is in part due to this decentralisation that Git is able to be so fast and versatile, as no query requires communication with another computer. Decentralisation reduces the risk of a single central breaking point for the system, removes the overhead of slow network connections, and allows the tool to be used even at points when a computer has no internet access, as changes do not need to be synchronised in real time.

Whilst this model confers considerable benefit, an issue which users of Git must deal with themselves is that there is no in-built authorisation mechanism, and no formal means of authentication in general. Git users themselves may choose to abide by a set of procedures which exist independently of the bounds of Git itself. Workflows typically fit into two categories: decentralised and centralised.

2.4.1 Centralised

Within a centralised workflow, a single central repository is used, and all members of a team are given access to read and write to it. They would typically pull a copy of the repository onto their local machine to make their changes, and push them back to the central server when they are ready to share their work with the rest of the team. As Git does not have a concept of user permissions, the workflow often resorts to "all-or-nothing" forms of authorisation; a user will either have access to update the entire repository, or none of it. It is possible to use a workflow which affords each user unique access permissions, but this requires third party software to achieve.

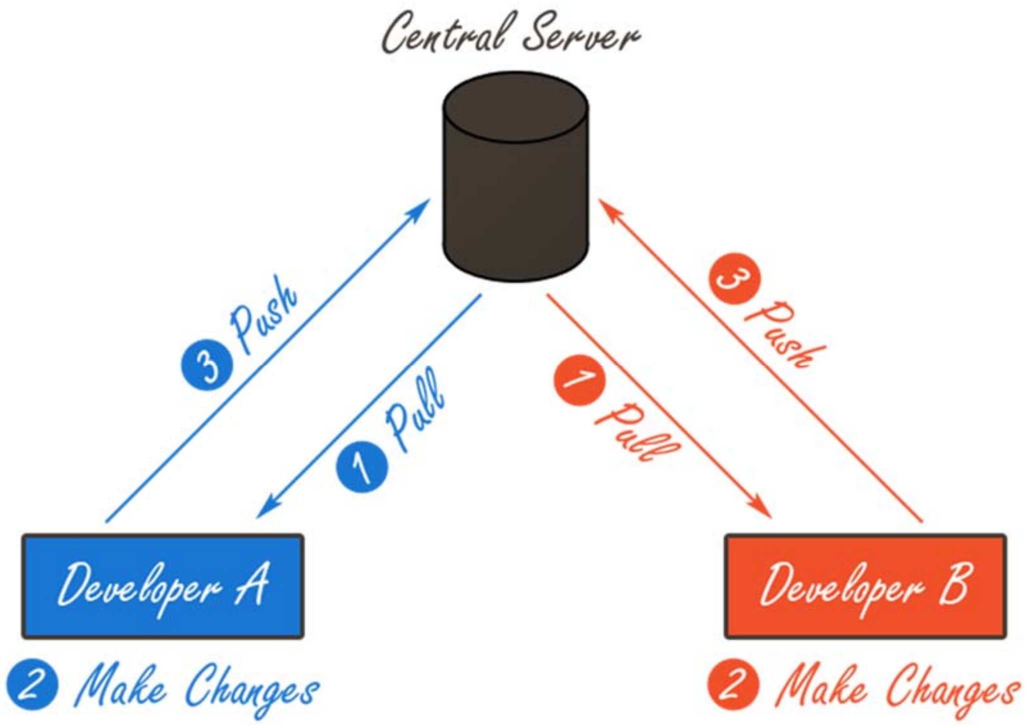

Figure 5: Two developers working independently via a centralised workflow

2.4.2 Decentralised

A decentralised workflow typically gives each contributor both their own public and private repositories – their private repository is used for development on their local machine, as with centralised workflows, and their public repository is used to publish those changes when they are ready. Users would not typically have write access to each other's public repositories, but they can read from them to review any changes. Discussion about which features or improvements are to be accepted will happen in some form of out-of-channel communication, typically email, and then a maintainer (or group of maintainers) will pull any accepted commits from the relevant public repositories, merge them together, and update the "official" public repository (often termed the *blessed repository*).

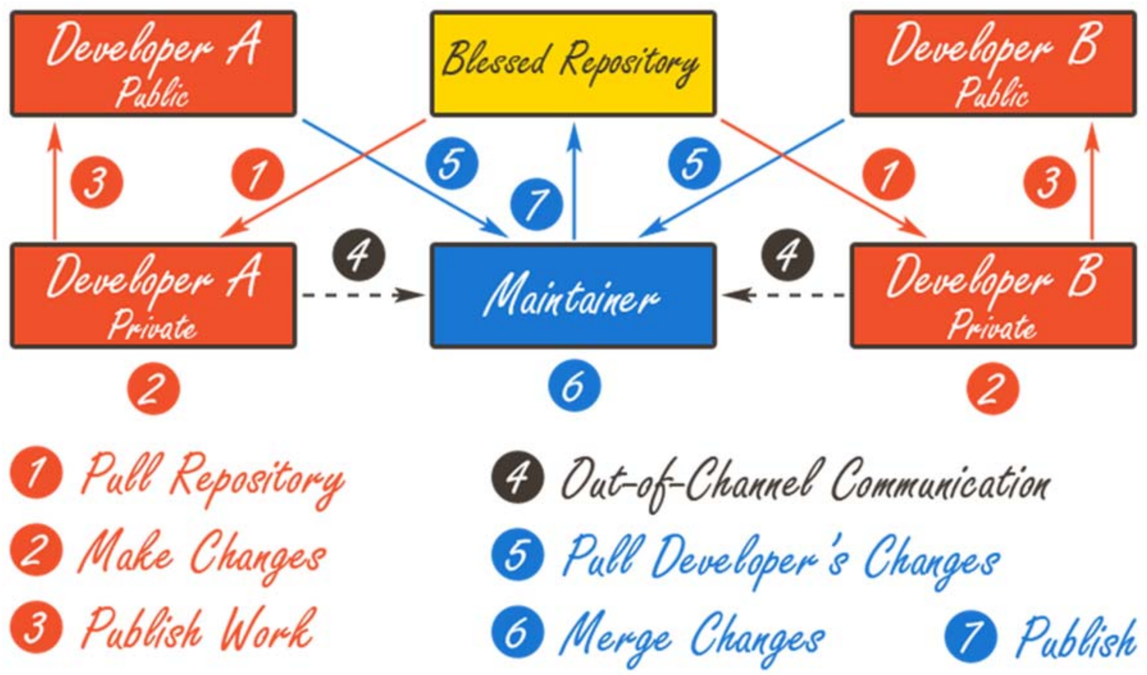

Figure 6: Two Developers working in a decentralised workflow via a maintainer

Both of these workflow categories are used for various Git projects, and are discussed in detail in the Git manual [9]. Within centralised version control software, such as Subversion, decentralised workflows are not applicable so they will always rely on some form of centralised workflow.

It is up to the users of the version control software to decide themselves which workflow is appropriate for their group. This requires agreement and extra overhead in the form of emails and other such communication. In a study on people's interactions within teams, it was found that on average, 75 minutes per day is spent on unplanned and informal communications [10] p.42. If the group fails to agree a suitable workflow, the project is without a form of validation or authorisation, as both of these are absent from Git and Subversion.

2.5 Potential Issues

When developing any software on top of Git's source code, or which makes use of a Git repository, there are several technical issues that one may need to be aware of, and mitigate.

2.5.1 Branch History after Merge

Git keeps track of branches by storing a pointer (known as a *reference*) to the most recent commit in that branch, known as the branch's *HEAD*. As such, any ancestors of this are implicitly thought of as being within that branch, as they can be reached via the reference to the branch's HEAD. It is possible for branches to overlap each other, for example, when two branches are merged together, the merge commit will be at the HEAD of both branches. As a result, it becomes impossible to determine which branch each strand of commits originally belonged to, as both can be reached via the HEAD of each branch.

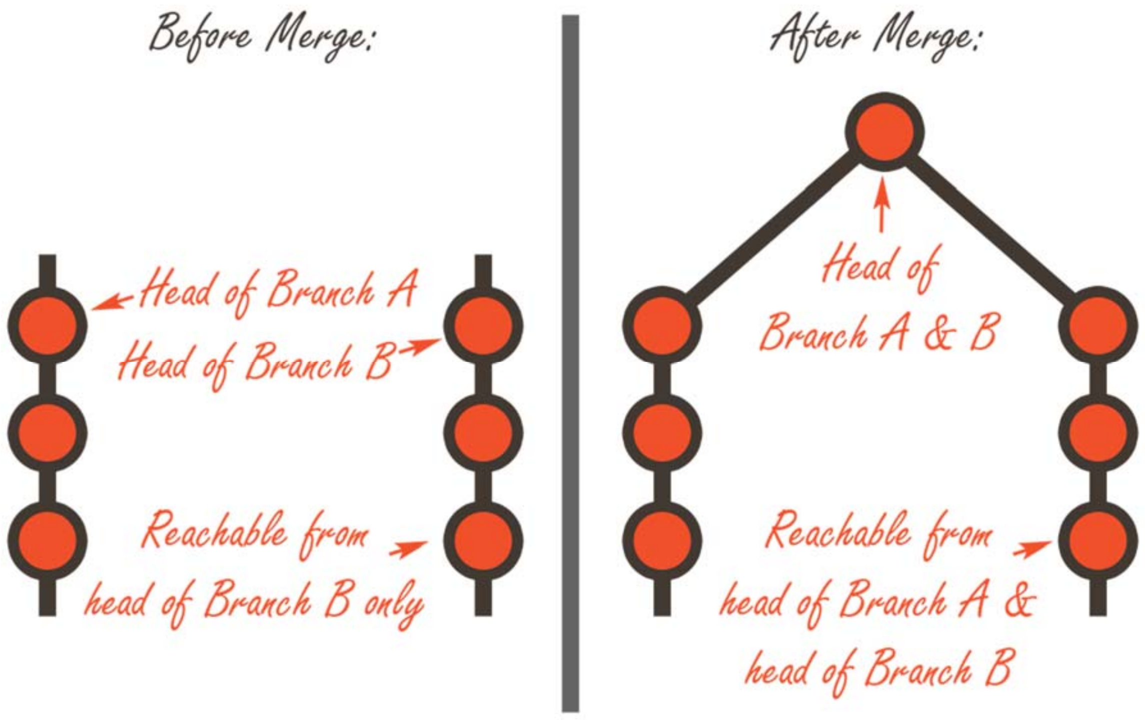
Before Merge:
Head of Branch A
Head of Branch B
Reachable from head of Branch B only
After Merge:
Head of Branch A & B
Reachable from head of Branch A & head of Branch B

Figure 7: Two branches merging into one, each adopting the other's ancestors

This functionality is used to make Git quick and extensible, however it results in a loss of history. If an application requires the retention of a commit's branch identity after a merge has occurred, this meta-data will have to be manually stored elsewhere.

2.5.2 No Forward References

Once a commit has been created in Git, it cannot be changed without invalidating or recreating it. As such, although each commit contains a reference to its parent, or parents, no commit can contain a reference to a commit to which it is itself a parent. This is not a drawback for the standard use of Git, as it is intended to be used to compare files and commits irrespective of history, or to work with a history working backwards from the current moment.

This, however, is a drawback should an application need to know which branch, or branches, a given commit is contained within. The only way to do this is to walk backwards from the HEAD of all branches known to the Git repository, until the commit in question is located.

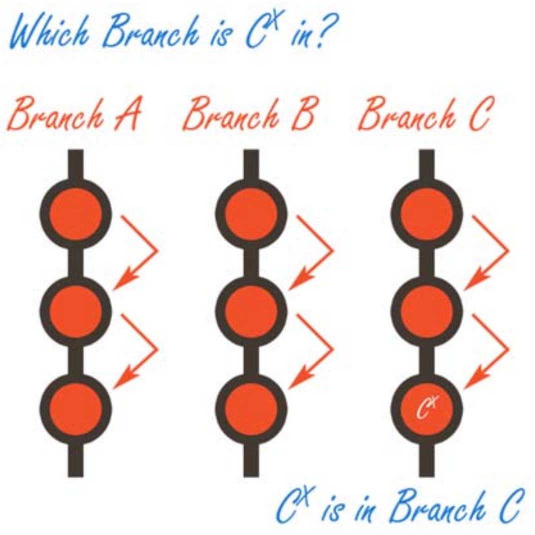

Figure 8: Searching for commit X by scanning from the start of each branch

2.5.3 Rebasing Destroys and Recreates a Branch

As discussed above, a commit cannot be changed without invalidating it. This is because a commit is identified by a hash of its contents. When rebasing a branch, Git must amend both the parent commit and the file-system snapshot for each commit in the branch. As this changes their content, and by extension identity, this can be thought of as deleting the entirety of a rebased branch, and recreating a new one in the new rebased position. Any reference to a commit in the branch prior to being rebased will become invalid and will have to be updated with a reference to the commit's new name.

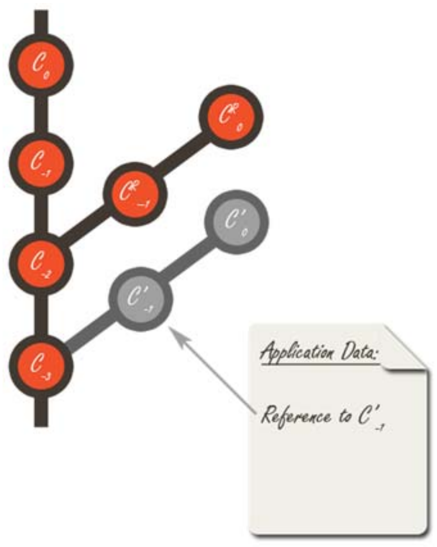

Figure 9: An application's reference to commit C'_{-1} is broken as it has become C^R_{-1}

Chapter 3

Competitive Research

As Git does not have any inbuilt concept of validation or authentication, many developers rely on third party software to define a set of workflow procedures, aid in code validation and enforce user authentication. This chapter reviews these systems and the techniques which may be used to automate user privileges within these.

3.1 Gerrit

A typical example of code review software is Gerrit [11]; this runs on a single server, on top of a standard Git repository. When a developer wishes to make a code proposal, they are able to push their changes from their local repository, which can be using a standard version of Git, to the central server for approval. Gerrit holds the commit in abeyance and allows the team to inspect, comment on and approve or reject the change via a web interface. When reviewing, developers may have the following options, depending on their permission level:

- "+2 Looks good to me, approved" – This approves the code immediately
- "+1 Looks good to me, but someone else must approve" – The code will be approved once another developer has also approved
- "0 No score"
- "-1 I would prefer that you don't submit this" – The code will be rejected if another developer disapproves
- "-2 Do not submit" – This immediately discards the code

Once approved, Gerrit will merge the changes into the main branch automatically, making this workflow pre-commit, as changes are not included until reviewed.

3.2 Wider Review of Peer Review Systems

There are several very similar packages to Gerrit, some of which follow a post-commit workflow, which allows teams of developers to review changes once made. These are summarised below:

	Workflow		Validation
Gerrit	Pre-commit	Centralised	Up to two reviews required
GitHub	Post-commit	Centralised	Line by line comments
Kallithea	Post-commit	Centralised	Line by line comments
Review Board	Pre-commit	Centralised	Variable number of reviews
RhodeCode	Both	Centralised	Variable number of reviews Line by line comments
Rietveld	Pre-commit	Centralised	Variable number of reviews Line by line comments
Crucible	Both	Centralised	Variable number of reviews Line by line comments
Stash	Pre-commit	Centralised	Variable number of reviews

Figure 10: Comparison of existing peer review systems [11] [12] [13] [14] [15] [16] [17] [18]

Using low thresholds for the required number of reviews is appropriate for corporate projects or small teams of developers, as each contributor is likely to be trusted and working towards the same aims and objectives as their peers. Users can be assigned differing permissions within these applications by an administrator, which is suitable for a manager-employee team structure.

However, for larger open source projects, this becomes impractical as although there may be users of higher standing within an open source community, there are rarely members with a strong authoritative status. Instead many decisions are discussed and agreed upon by the community at large, as outlined in the decentralised workflow above.

It is clear from Figure 10 above that all of the current code review software relies on a centralised workflow. There are two important implications of this: firstly, users within the system must delegate trust to a central authoritative server, and a person, or group of people, must agree to manage it. This may be unsuitable for open source projects as it creates a barrier to a peer's ability to confirm that the server is handling votes and reviews appropriately. There is also a cost overhead of running a server for a project to which many developers may wish to contribute. For example, in their 2012-2013 financial year, the Apache Software Foundation, which supports open source projects, reported a total expenditure of $501,556 of which $353,000 was infrastructure costs including hosting source code, mailing lists, bug tracking systems and collaborated software [19].

Secondly, a centralised system of this nature interferes with Git's ability to be speedy and usable offline, as reviewers must connect to a web interface to comment and cast their votes.

There is currently no software which manages the alternative: a decentralised workflow. This is the space in which this project aims to investigate.

3.3 Privileges within Peer Approval Systems

All of the current peer approval software analysed above allow administrators or managers to grant users differing privileges and permissions within a project. This, however, is inapplicable for open source projects, as there normally is no rigid management structure in place. In theory, within an open source project all developers are regarded as equal, and all may contribute to all aspects of the work, to be reviewed by the group. This is only possible within the decentralised workflows discussed above because there is a central maintainer. This figure need not be authoritarian, but often use human judgement to separate serious suggestions from weaker ones, and may be influenced by the proposer's reputation within the community.

This leads us to the issue of reproducing the administrative effectiveness of a maintainer in a truly decentralised system in which there is no such agent. The Team Software Process states that "*all … aspects of the development process should be based on data wherever possible*" [2] p.14, in this spirit, this section will explore the concept of quantifiable reputation, which can be assessed, gained and revoked automatically, and systematically, by a decentralised system.

The concept of reputation has been explored before for commercial purposes online. For example, Ebay Inc, a large peer-to-peer sales platform, uses buyer feedback to decide a seller's reputation [20] [21]. This impacts sellers in two ways: as their reputation is publicly available to all who would do business with them, having a low reputation may serve as a deterrent for potential buyers, or by contrast, having a high reputation may sway a buyer's decision in that user's favour. Ebay also employs automatic checks such that when a seller's reputation reaches a very low score, or a specific complaint is levied against them, they may be banned from using the site, either permanently or for a probationary period of time.

This introduces the possibility that negative factors may be as important in deciding a user's trustworthiness as positive ones. As a result, any decentralised Version Control Peer Review system may need to take into account a user's poor proposals and reviews in some way.

Sites such as those managed by Stack Exchange [22] use a similar system to that of Ebay, where users may rate another user's questions or answers either positively or negatively. Should a user receive a large quantity of negative reviews in a small period of time, all of these votes will be automatically reversed by the system. This requires a body to perform such a check, which may be difficult to achieve in a distributed system. Stack Exchange also uses a peer's reputation to decide their privileges within a system. For example, users are not allowed to down vote other users until they have achieved a certain standing themselves. As a user gains further reputation, they are trusted with more privileges and powers.

A completely different approach is used by Facebook, and many other large social media sites, such as Twitter, YouTube and Google+. Facebook does not take negative factors into account, however by default each item is initially viewed with lower trustworthiness. As such, Facebook will only show a given post to a subset of the users who have subscribed to see it, and only if it is received positively is it shared to a greater subset of people. In this way, users are presented with less information to process themselves as they are only shown content which Facebook has either determined they have a high probability of liking, or a few select pieces which they are inadvertently moderating for the community as a whole. This introduces a distributed structure within the system, however it relies on both there being a central system to share out the content evenly, and assumes that the set of users which will review a given piece of content will engage responsibly and respond to it in a manner which is representative of the whole population.

For greater powers, such as full site or user moderation, Wikipedia [23] and Stack Exchange [24] sites use conventional elections to select from candidates. Nominations are normally open to any registered user on the site, and their reputation may, or may not, be taken into account by the electorate when voting. This suggests there may be certain privileges which are either too specialised, or too important, to be granted without human review.

Chapter 4

“Legit” Specification

This project evaluates the benefits of a decentralised peer approval model by developing software to enforce this workflow. It is based upon Git version control software, and is called *Legit*.

4.1 Overview

Legit manages a Git project by introducing the concept of proposals and reviews, in a similar form to pull requests, and requires that changes be voted on before they are merged into the main branch of a project. A user may be automatically granted additional powers or permissions within a Legit project as their ”reputation” grows – this is based upon their activity on the project to date, including such things as the number of accepted proposals they have made.

As all users have an equal opportunity to vote and to make proposals, including administrative proposals such as solving merge conflicts, a Legit project does not have the same requirement for a maintainer as traditional decentralised Git workflows do. There is no administrative overhead, such as a central server which is required in current peer approval software, or a blessed repository which has previously created a centralised aspect within most decentralised Git workflows.

All voting, reviewing, proposing and commenting is performed within Legit, meaning project members no longer need to rely on informal and out-of-channel communications such as emails. This aims to make the decentralised workflow more accessible to open source projects by reducing the disciplined approach that is traditionally required.

4.2 Legit Functional Specification

4.2.1 Introduction

Each developer within a Legit project has a *reputation* associated with them. This is calculated from a record of their activity within the system, specifically the number of accepted, rejected and pending proposals they have made, the number of reviews they have given, and the number of those reviews which either agreed with, or disagreed with, the ultimate result of the vote. These variables are recorded for everyone, and the weights of each one when determining if a user has appropriate reputation for a given task is configurable.

Legit allows a branch to be set as *locked*, meaning users cannot normally[6] commit directly to it; instead they must submit a *proposal* (see below) for peer review which, if approved, will then be merged into the locked branch.

4.2.2 Proposals

Any user may make a *proposal* if they have the appropriate level of reputation within the system. This is a branch of commits which is based upon a commit in a locked branch or a commit within another proposal. A proposal shall also have a description of the changes along with the proposer's name and time of proposing, and can be one of the following: normal, merge, fix or extension:

- *Normal* proposals must always be directly based upon a commit in a locked branch.
- *Merge* proposals are required by Legit when an approved proposal cannot be merged automatically into the locked branch.
- *Fixes* represent an alternative to another proposal, normally for use in bug fixing, problem solving or optimisation. Of a proposal and any fix proposals of it, only one can be accepted.
- *Extensions* must always be based directly upon a commit in another proposal. An extension, if accepted, will only be merged into the locked branch if the proposal, or a fix of that proposal, on which it is based is also approved. In the case that an extension's parent proposal is replaced by a fix proposal, Legit will automatically attempt to rebase the extension upon it, and submit that as a new merge proposal.

4.2.3 Review of Proposal

A peer with the appropriate level of reputation may review a proposal. This consists of a comment about the proposal, the name of the reviewer, the time of the review and an *accept* or *reject* vote. A peer may only vote once, and is not allowed to edit another peer's review.

[6] Users of a sufficiently high reputation may be granted permission to commit directly to a locked branch.

4.2.4 Accepting a Proposal

In order for a proposal to be *accepted*, it must obtain a configurable amount of net accept votes. Net accept votes are defined as the total number of accept votes less the total number of reject votes. On acceptance, Legit shall attempt to automatically merge the proposal into the locked branch it is based upon unless it is based upon another proposal which is still in voting or has been rejected, in which case it shall be held in abeyance until its parent proposal, or a fix of it, is approved.

When attempting to merge an accepted proposal into the locked branch, if this cannot be done automatically, the proposal is held in abeyance until any user submits a merge proposal for it.

4.2.5 Rejecting a Proposal

A proposal which fails to secure the required number of votes after a certain amount of time, or which receives a threshold number of negative votes is deemed to be *rejected*. If desired, a client computer may delete a rejected proposal, along with all commits that are associated with it only, however the associated meta-data must be kept intact.

4.2.6 Accepting a Push or Pull

When a legit repository is updated, the receiving client should check all new commits to ensure they have followed the defined voting rules.

4.3 Legit Workflow

As Legit contains all the administrative tasks involved in a project within the repository itself, it is able to use a decentralised workflow which does not contain the complexities or overhead of a maintainer-based workflow. Figure 11 below shows how Developer A and B may work in a truly independent fashion without the need for any centralisation.

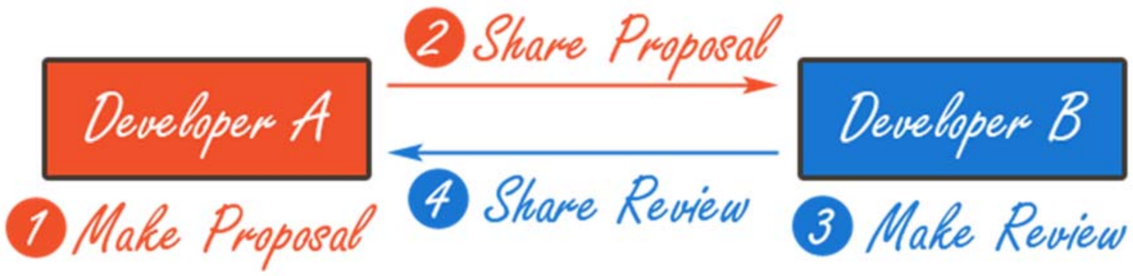

Figure 11: A developer making a proposal which is reviewed by another developer

This, when compared to Figure 6, plainly shows a cleaner workflow, with several important advantages: a Legit project does not rely on email exchanges or other such communication, meaning a formal audit trail can be kept for each proposal and any comment or proposal is much less likely to be missed or miscategorised. The financial and administrative overheads of running, checking and managing a mailing list and blessed repository are completely removed from this workflow as there is no need for a maintainer or central repository. The barrier to become a contributor is also reduced, as there is no longer a requirement to run a public repository, or subscribe to long mailing lists.

For ease of use, a central server *may* be used to create an accessible point for all developers to obtain and update the project, but that would serve as convenience only; it is not a requirement of Legit that it is there. This is shown below.

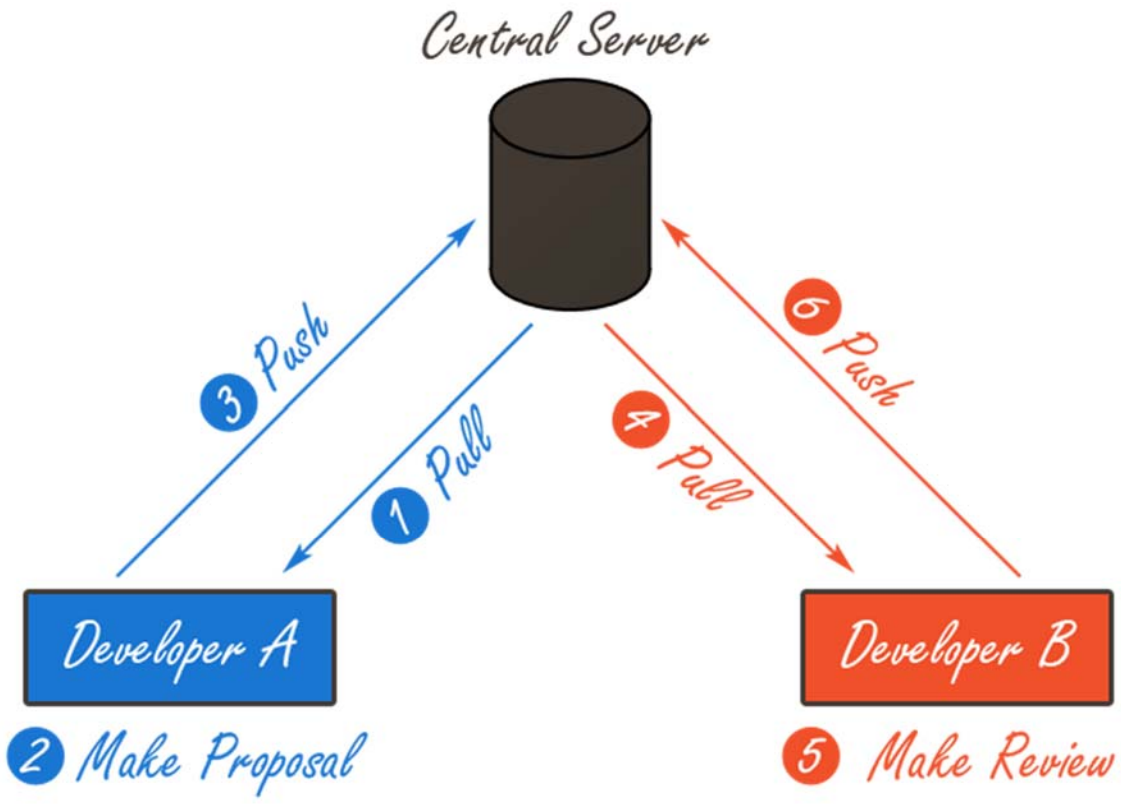

Figure 12: A developer making a proposal which is reviewed by another developer via a central server

This workflow appears very similar to a centralised workflow, as shown in Figure 5, however it is important to note that in a centralised workflow, the central server is normally considered to be a trustworthy source, whereas this is not the case within Legit. A central server, if one should exist, would act solely as a mirror for the distributed project; it can be run by anyone, even those otherwise uninvolved in the project, and clients themselves are able to verify the integrity of anything downloaded. As a mirror server has no authority within a Legit system, there may be many, which can be independently run. This gives great flexibility, as a distributed network of this scale would normally require central management.

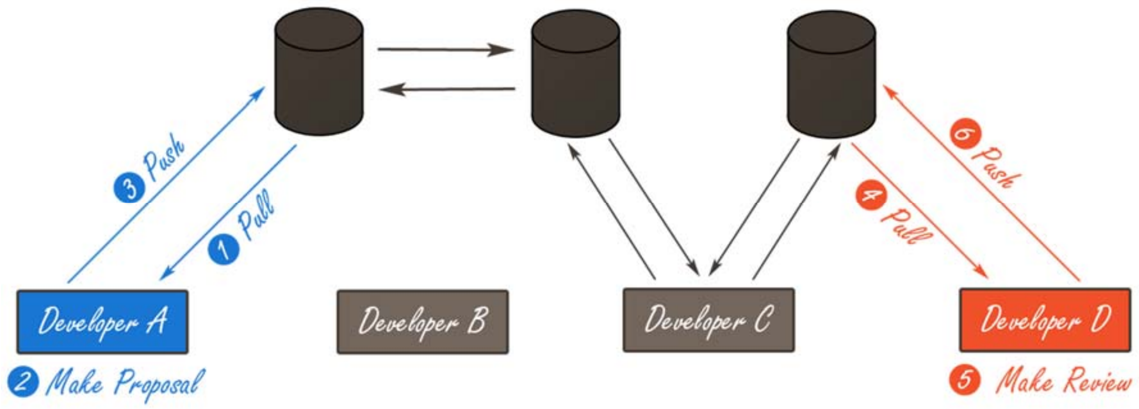

Figure 13: Developer A proposing a change, which is reviewed by Developer D via a distributed network

Chapter 5

Design Discussion

As we have seen, Git is a feature rich piece of software, allowing multiple branches to exist independently before being merged into one another. The Legit specification adds a set of new features on top of this, which both deals with the inherent complexities of a Git project, and adds to them. This chapter discusses many of the functional and technical design choices.

5.1 How to Decide Reputation within Legit

Within a Legit project there are a variety of actions for a user to perform, any or all of which could feasibly contribute to their reputation in some way. Below is an analysis of the benefits and issues associated with some of these quantifiers.

5.1.1 Vote or Decision Based Reputation

It is clear that good proposals should normally contribute to increasing a user's reputation. There are two ways of doing this: the reputation gained from an approved proposal could be proportional to the number of votes it receives. For example, a proposal which is approved by 10 users would result in less reputation for the proposer than if it had been approved by thousands of users.

Alternatively, simply the accept or reject decision, irrespective of number of votes could contribute a constant score to a user. That is, a proposal accepted by 10 votes would gain a user the same amount of reputation as if it had been accepted by thousands of votes.

Legit uses the second proposal for two reasons, one philosophical and one practical. In philosophical terms, a code suggestion's value should not come from how popular or interesting it is, but by whether or not it is deemed to work. Secondly, a system in which the number of votes contributes directly to reputation is open to abuse. Users may vote to accept proposals which have already been accepted to appear as though their reviews are correct, and users may ally together, voting for each other's accepted proposals, boosting their reputation without contributing anything new to the system.

5.1.2 Scores for Proposing and Reviewing

It is logical that good proposals should contribute positively to reputation, however, as discussed in section 3.3, it may be appropriate to penalise poor proposals to discourage users who may wish to spam a project with inappropriate proposals, or who may attempt to insert malicious pieces of code. A disadvantage to this policy is that it risks stifling creativity if the penalties are too high.

Sites such as Stack Exchange also take into account the way in which a user reviews others. The simplest form is to give a user reputation whenever they make a review of a proposal. This would be beneficial as it would motivate developers to read the proposals of others. Increasing the number of developers who take the time to check the code of others may reduce the likelihood that errors are made, either in the form of good proposals being missed, or incorrect code being accepted. This also serves to unify developers, as reading snippets of code produced by others will give them a better understanding of the system should they wish to make a proposal themselves, and will aid them to pick up the project's chosen coding standards and traits.

However, this can clearly be abused should a user decide to review everything for the sake of improving their score – this would, in fact, harm the system, as should they vote to accept every proposal, they would risk allowing code that is malicious or defective into the locked branch.

Conversely, a user may attack another by voting against all of their proposals, in an attempt to damage their reputation. Stack Exchange combats this by penalising any form of negative vote, in an attempt to encourage users to use the feature responsibly and not in bulk. This is an adequate solution for Stack Exchange, as a 'question and answer' forum, down-voting is not essential and only serves to reduce a post's visibility on the site. However, within a coding project, it is equally important to assess and reject poor code as it is to accept working code. As such, it may not be ideal to add a penalty, as this may discourage peers from ever voting to reject a proposal, even when it is appropriate to do so.

From this we may conclude that both negative and positive votes should contribute to reputation, when they are cast responsibly. In order to determine whether a vote was responsible, or indeed sensible, we can assess whether it is in line with the votes cast by the majority. For example, should a proposal be approved by popular vote, it may be suspected that the minority voting against it were doing so maliciously. It is important to recognise, however, that it is not always the case that disagreement implies malice. This strategy may run the risk of encouraging a user to vote with the cohort, rather than using their own opinion. As a result, many undecided peers may be assimilated into the majority, forcing a single viewpoint for the project as a whole. Depending on the size and nature of the project, this may or may not be a good thing.

Each of these four points above - to penalise poor proposals, to reward all reviews, to penalise poor reviews and to penalise minority reviews - have real advantages and disadvantages, which may be more or less relevant depending on the specifics of the project. For example, for a small team, abuse may be easier to spot and deal with outside the bounds of Legit, meaning users may prefer a solution which encourages decision making rather than abuse protection. As the size of a project grows, automatic abuse protection, and ensuring the views of the majority are largely equal, may become more appealing. Therefore, Legit supports all the above schemes, and allows their importance to be enabled, disabled and weighted via the project's configuration.

5.2 Meta Data Storage Schema

The first technical issue to be considered with Legit, is the storing of the added meta-data which accompanies any or all peer approval systems. In Legit's case, the following information needs to be stored:

- Names, details and credentials of all collaborators on a Legit project
- Details of all proposals, and a record of those which are currently open for review
- All reviews, including the votes cast
- Statistics for each user on their activity in the system

There are three main storage methods which could be used to store meta-data within a Git repository. These are: tags, a directory in the project's root directory, or separate orphaned branch. The advantages and disadvantages of each are analysed below.

5.2.1 Tags

Git allows tags to be created to annotate commits and files, adding meta-data to the repository. It seems a logical starting point, therefore, to investigate their usefulness for Legit. As tags are stored separately to the project's files, the working tree is kept free from files which may otherwise confuse users.

However, tag objects do not exist under version control, making them difficult to edit once created. Git will not keep records of any changes made to tags, meaning there would be no audit trail for votes and discussion. Sharing edited tags would be a challenge as Git does not have a mechanism to merge differing versions of tags. As a result, although tags are a useful feature for creating static annotations, they are unsuitable for the dynamic meta-data that Legit requires, such as vote counts and statistics which must be constantly updated.

5.2.2 Meta-Data Directory in Branch

Keeping ever changing records is a critical requirement of Legit, and in a voting-based system, an audit trail is preferable. Both of these requirements can be fulfilled by Git itself if the meta-data is stored within its working tree. This can be achieved in one of two ways:

5.2.2.1 Directory in Root of Locked Branch

All meta-data can be stored within a directory in the root of the locked branch, meaning any changes would need to be committed to the branch. When a proposal is made, a new sub-directory is added to contain the proposal and review meta-data, with a pointer to the branch, or commit at the head of the proposal in question. When users make a review, this would be committed to the end of the proposal, making the commit log a history of both code changes and the discussion.

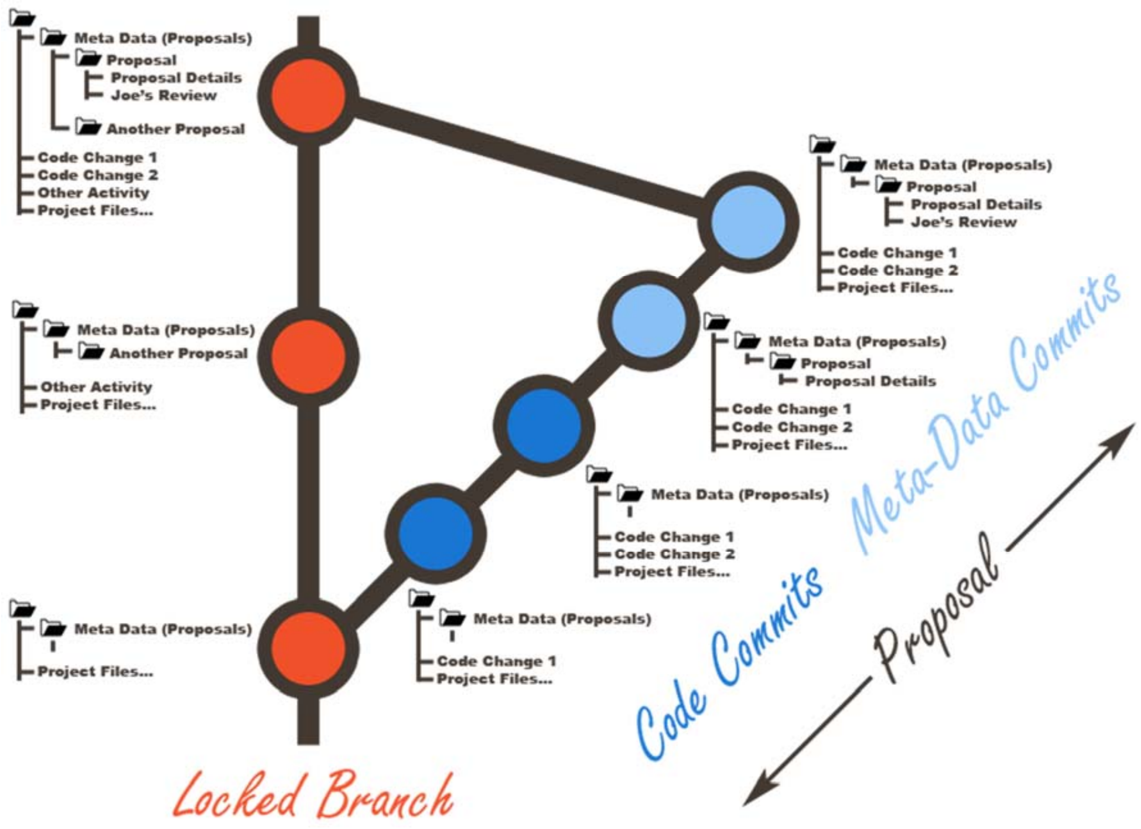

Figure 14: A proposal, made up of two commits, is created, reviewed, approved and then merged into the locked branch

A possible disadvantage of this is that meta-data would be stored in the same space as the code under source control – this may interfere with development and could be confusing to vanilla Git users.

More seriously, this only allows the recording of reputation associated with accepted proposals; as rejected proposals are not merged into the main branch, all record of them ever having being made will be missing from the locked branch, meaning a user will not get any negative reputation for poor proposals, or reputation for any reviews given on proposals which were ultimately rejected.

5.2.2.2 Specialised Directory in Root of Proposal Branch

As an alternative to the schema above, a specialised meta-data directory could be placed in the root of the working tree when a proposal is made, containing only proposal-specific data. When a user wishes to make a review, Legit would place this in a file within this directory and commit it to the proposal branch.

When merging an accepted proposal back into the locked branch, Legit need only merge the commit at the head of the code changes, leaving the meta-data commits above them as a "dangling" branch. As a result, a proposal branch would contain an audit trail of both code changes and discussion, however, as only the code is merged into the locked branch, it is free from all meta-data and meta-data related commits.

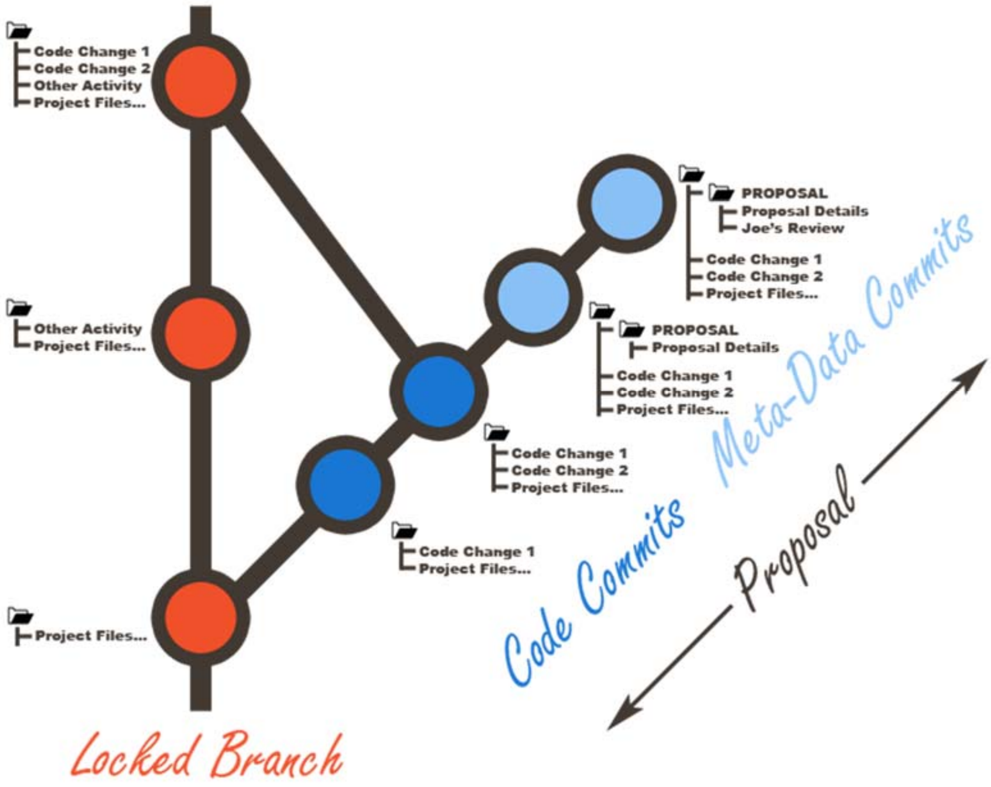

Figure 15: A proposal, made up of two commits, is created, reviewed, and approved. Only the code commits are merged into the locked branch

A drawback to this technique is that these branches are stored in an unordered and difficult to access manner, and it requires keeping an active branch for each proposal, even after they have been accepted or rejected. Using multiple branches does not allow for the storage of statistics, which are required to calculate a user's reputation, meaning Legit would have to compute this value whenever it is required.

5.2.3 A Separate Tracking Branch

We have seen that Legit requires the version control which a branch can offer, however an optimal situation would allow for the storage of branch-independent information. For this reason, Legit stores all meta-data in a separate branch – this keeps Legit's information completely separate from the working tree, making its presence as transparent and unobtrusive as possible.

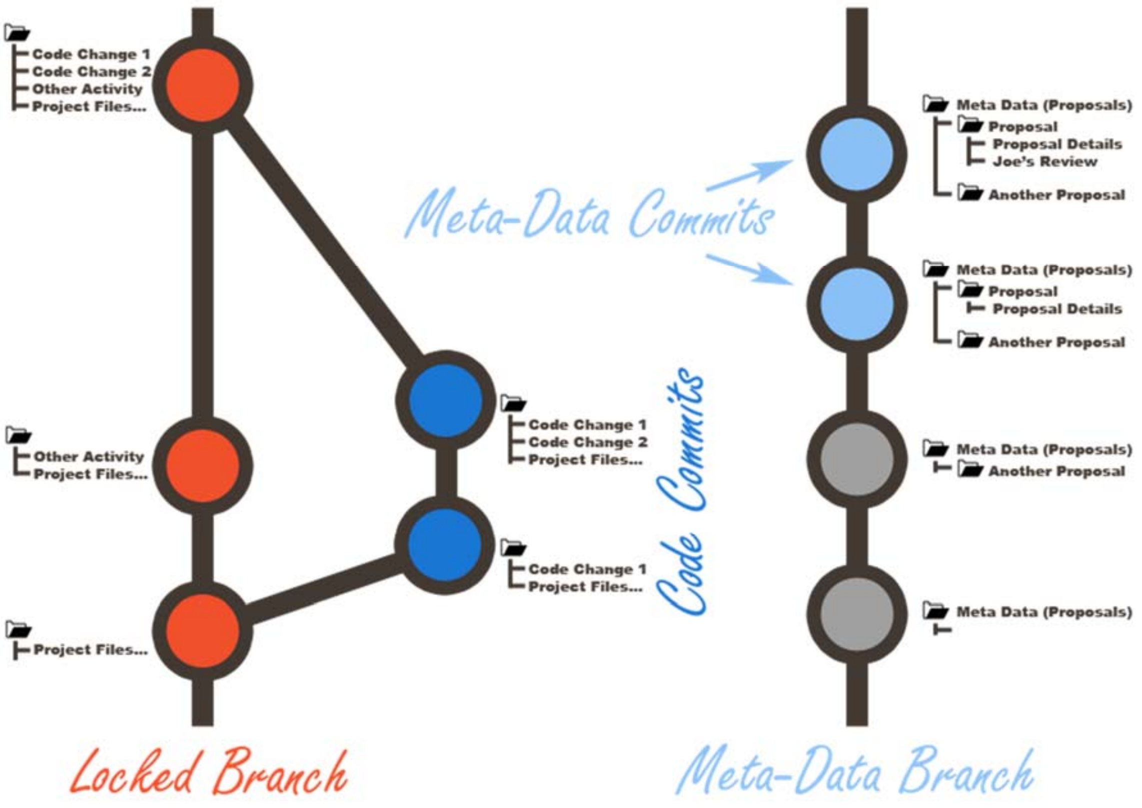

Figure 16: A proposal, made up of two commits, is created, reviewed and then approved

5.3 Finding the Start of a Proposal

Within the meta-data associated with a proposal, the original specification saved only a pointer to the head of a proposal, leaving the starting point to be worked out by any client working with it. This is appropriate for normal proposals, as these are based directly on a locked branch, so the starting point can be determined by walking back along the proposal branch, until a commit within the locked branch is found.

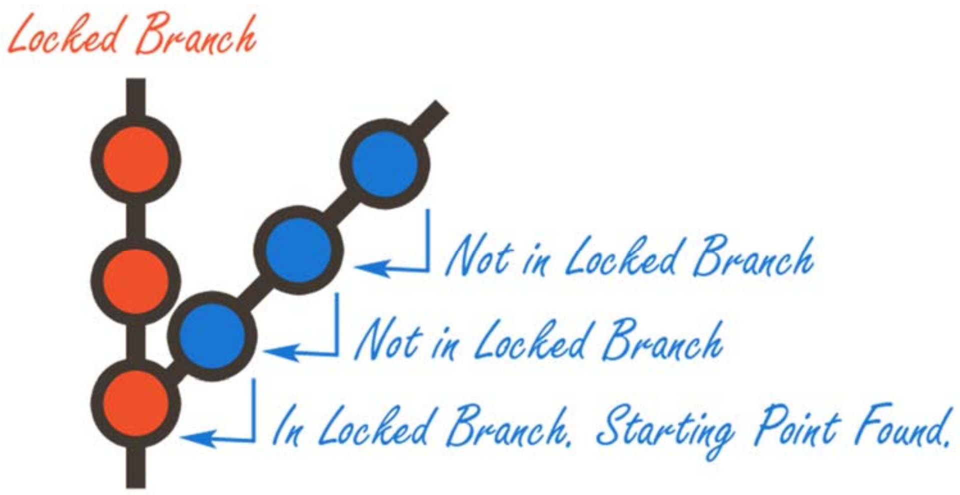

Figure 17: Searching for a proposal's starting point by walking along it, until the locked branch is found

This technique, however, is relatively slow, as finding which branches a commit is contained within is a complex process (as discussed in section 2.5.2) and this must be done on every commit until the locked branch is found.

As extensions and fixes do not need to be based on the head of the proposal they are based upon, not storing a proposal's starting point can lead to ambiguity. For example, should a proposal have an extension based upon it, and a second extension based upon a commit within that (as shown in Figure 18), it is unclear which extension builds upon the other.

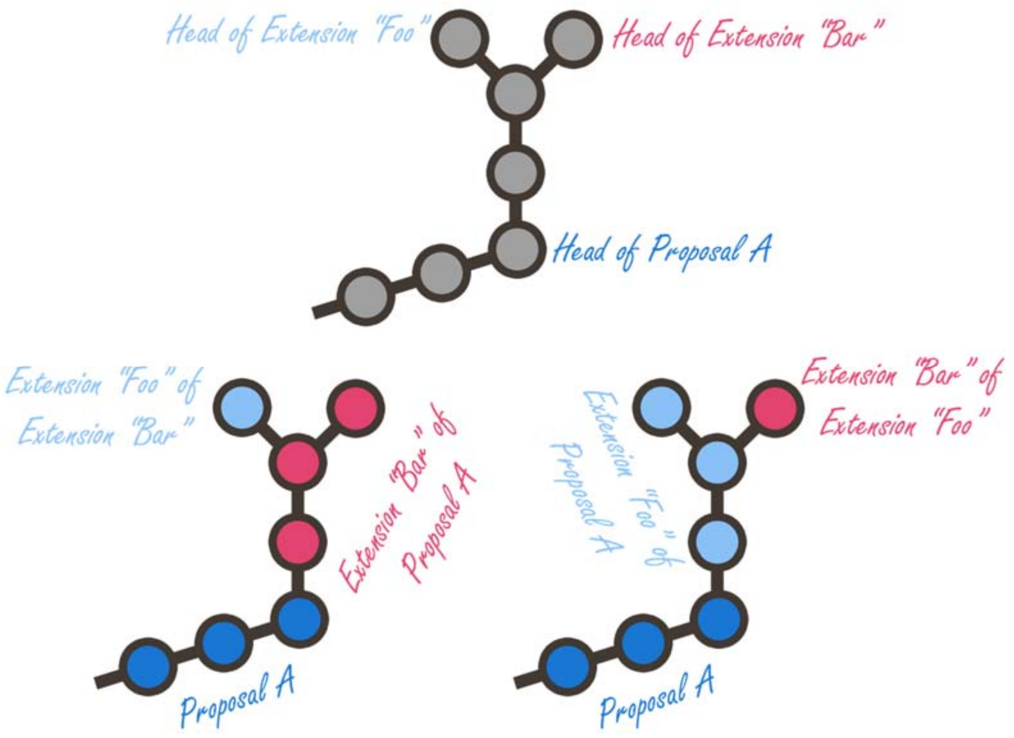
Head of Extension "Foo"
Head of Extension "Bar"
Head of Proposal A
Extension "Foo" of Extension "Bar"
Extension "Bar" of Proposal A
Proposal A
Extension "Bar" of Extension "Foo"
Extension "Foo" of Proposal A
Proposal A

Figure 18: A tree structure annotated with known information (shown above), and two interpretations of this (shown below)

The solution to this, fortunately, is simple: at the time of proposing either of these extensions, the proposing client is certain to have known the true state of the system. For example, if we are to assume the interpretation shown on the left of Figure 18 is the correct one, when "Bar" was created, "Foo" could not yet exist. As such, it is clear "Bar" is an extension of "Proposal A". Similarly, when "Foo" is then proposed on top of "Bar", that client is plainly aware of "Bar"'s existence (This is shown in Figure 19).

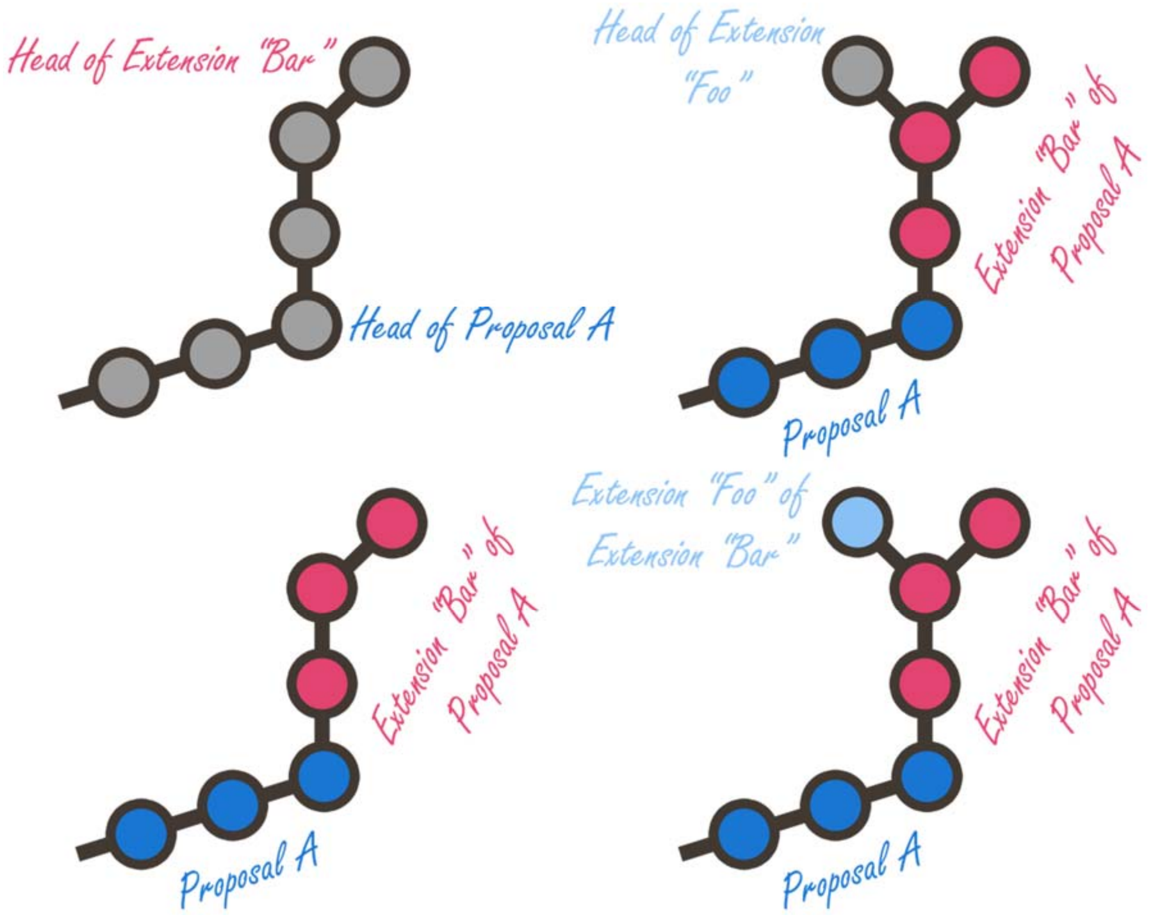
Head of Extension "Bar"
Head of Proposal A
Head of Extension "Foo"
Extension "Bar" of Proposal A
Proposal A
Extension "Bar" of Proposal A
Proposal A
Extension "Foo" of Extension "Bar"
Extension "Bar" of Proposal A
Proposal A

Figure 19: Without the presence of "Foo", "Bar" clearly extends "Proposal A" (shown left). With this knowledge, the later addition of "Foo", means it extends "Bar" (shown right)

As such, Legit's technical specification was amended to state that when a proposal is made, both its head *and* starting point must be stored in the meta-data. This not only clarifies all ambiguous cases, but allows clients in the future to work more speedily with the proposal, as they are no longer required walk along the branch to find its branching point (as discussed above, this is a relatively slow process).

Chapter 6

Project Management and Testing

As this project is a single person team, it did not utilise a strict Agile method such as Scrum, which is most useful for organising teams of developers. Instead, the Spiral model was chosen as it affords the developer ample opportunity to assess and review changes whilst progressing, without enforcing too much administrative overhead, or a strict timetable. The project's timeline is described in this chapter (and is illustrated by the Gantt chart in Appendix D).

6.1 Risk Assessment

A risk assessment was carried out prior to the start of any development or research. This highlighted several risks, but only one had a serious priority score. This did occur, however the planned mitigation was successful in addressing this. The remaining risks informed the design of the project schedule and influenced the choice of tools and primary programming language. Please see Appendix E for the full risk assessment table.

6.2 Initial Research

The first stage was research into current code review and peer approval systems, such as Gerrit and GitHub. It was quickly concluded that almost all of these supported Git, many exclusively, and all of these were centralised. As a result of these findings, a broad requirements plan was drawn up, focusing attention on a decentralised system, and research was done into the market share of Git, to determine if it would be appropriate to base a system exclusively upon it.

6.3 Proving Viability

As no other decentralised peer review system was found in the research, it was important to prove its technical viability early on in the project. An abstract meta-data system was modelled according to the schemes detailed in section 5.2. These were investigated by creating mock locked branches, code proposals and the appropriate meta-data files, and manually entering Git commands to interact with and store these.

It was concluded that the project is indeed viable, and the separate tracking branch storage scheme was chosen. This led on to creation of a skeleton technical specification detailing the specifics of meta-data storage.

6.4 Creating Reputation

Following on from this, further research was carried out into systems which are able to use truer interpretations of "peer review" than those tools based upon code development. This lead to the concept of reputation-based privileges, and a full functional specification was created to include this.

6.5 Prototyping

To further understand the technical requirements involved in peer approval, and quantifiable reputation, prototype scripts were developed to aid in the formalisation of the technical specification.

6.6 Coding Development

As Git is made up of a series of atomic commands, Legit followed the same philosophy. These facilitated the splitting of the development into smaller more manageable sections.

6.7 Testing

Fortunately, the Git project contains its own test library; as Legit is built on top of the Git source code, the project was able to use this. The output adheres to the Test Anything Protocol (TAP), which defines the output of testing scripts, such that they may be run by a language and application independent test harness [25].

This pre-existing harness was particularly useful when ensuring Legit functionality did not interfere with standard Git functions. For example, Legit amends Git's default commit behaviour to prevent users committing directly to a locked branch – running Git's tests along with Legit's ensures that pre-existing functions, of which many rely on commit behaviour, are unaffected by this.

6.8 Report

The report was written iteratively as the development of the research, specification and software progressed. Typically, as aspects of the specifications were determined and refined, notes and diagrams detailing the thought processes behind these revisions were made in rough at the time, and rewritten a few weeks later. These neater pieces formed the basis of the design discussion in Chapter 5. Similar notes were taken during the research phases of the project, either describing findings in full, or simply making a note of the relevant source to be written up at the end. A "storyline" was written early on, to ensure all pieces followed a logical narrative structure. Clearly, time was reserved at the end to finalise and edit this.

Chapter 7

Critical Evaluation

Against its specification, the implementation of Legit performs well; this project has succeeded in creating a distributed peer review tool, where before there were none. In developing and testing Legit, we are able to evaluate the specification itself, to determine if it is a practical solution for modern software development. This chapter details the main three advantages found, and two issues which may require further attention or work.

7.1 Advantages

7.1.1 Accessibility of Open Source Projects to an Outsider

Currently, many large scale open source projects follow a decentralised workflow. As discussed in section 2.4.2, this involves out of channel communication, in most cases this comes in the form of a mailing list, which new users must subscribe to and email when they are ready to submit their changes. As emails have no inherent structure in themselves, but instead rely upon a user to define, or derive, some form of structure from them, this may create a barrier to entry for new developers. Should one subscribe to a mailing list of a team of undisciplined developers, interpreting the conversation and status of issues may be a difficult challenge. Conversely, joining a very disciplined team may prove an intimidating challenge, if the mailing list's terms of etiquette are not made clear, or are complex.

This is a less significant issue for Legit, as it does not rely on the user's ability to communicate in a structured manner. Instead, the structure is defined for the team.

7.1.2 No Middle-Management Layer

Although this paper has largely focused on open source and informal team projects as the target for Legit, it has a viable application within corporate environments. Where a business project finds itself requiring a middle-management layer, these employees are likely to be filling a maintainer-esque role within the project, by ensuring that the code written by the developers below them is of an acceptable standard and fulfils upper management's specification. When using Legit, the need for such a position is reduced, as the system empowers the developers to handle administrative tasks on their own.

Not only would this be a clear cost benefit for the company as a whole, it may have the potential to improve the working environment for the lowest level developers, by making them feel more part of a team responsible for their own work. Humphrey et al. assert that "*knowledge workers are best enabled to do creative work when they manage themselves*" [2] p.14. The extent to which this is true, and the positive effect this may have on their productivity, is testable and may be the subject of future work in the much broader area of research into creativity and motivation within large corporate teams.

7.1.3 Verification

As discussed in section 2.4 above, remote repositories are normally thought to have been vetted outside of the bounds of Git, in the case of most decentralised workflows, or to be implicitly trustworthy, as is the case in many centralised workflows. Because of this, Git does not have any inbuilt form of validation, or a way to define one's own validation behaviour, when performing a pull. Legit changes this paradigm, as all nodes within a Legit network are thought to have equal standing. Verification is performed within Legit on a pull *as well as* when receiving a push; this eliminates the need to delegate trust to, and manage, any central point. This results in an inbuilt robustness to Legit, which is both appropriate for many important open source projects, and structurally missing from Git.

7.2 Disadvantages

7.2.1 Security

By default in Legit, all proposals are synchronised when a user performs a push or pull. This gives the potential for attackers to plant malicious code on any developer's personal computer. In response to this threat, Legit does not automatically run any unapproved code. To aid a user to identify a threat before executing any code themselves, Legit collates the contents of the proposal together with all comments from other peers. It is a notable drawback of the system, however, that users must be aware of this risk.

7.2.2 Joining a Legit Project

Legit currently allows a new peer to be added to the system by running the command "legit add-contributor". The details of who would be responsible for adding new members, or what procedures and checks may be in place for this, are not defined within Legit's functional specification as presented in this report. In Legit's current form, it is possible that this process would work best with a degree of centralisation, however this could be addressed in a later version of the software.

Chapter 8

Conclusion

The Legit software code meets the specification as defined above – it *is* a distributed system for peer review; in this sense, therefore, the project has been a clear success. This chapter will discuss the benefits of future work and the concluding thoughts about the possibility of using such a system in practice.

8.1 Further Work

8.1.1 Extending Communication and Bug Tracking

The Legit system performs well within its defined remit: managing the discussion and administration of code proposals. An obvious extension to this, therefore, is an investigation into the feasibility of expanding this remit, to cover other important aspects of the software engineering process, such as discussing bugs and tasks, and assigning these to specific people.

8.1.2 Broader Voting

Currently, Legit only facilitates voting on proposals, however it would likely be a benefit to allow users to vote on a wider variety of things, such as the aims of a project, to elect project administrators as Stack Exchange and Wikipedia do, or even to add new contributors into the team.

8.1.3 Truly Separate Tracking Branch

The use of the tracking branch as opposed to the other meta-data storage methods discussed in section 5.2 allows for reputation to be easily shared between locked branches within a project. A potentially useful addition might be to develop a Legit system in which the tracking branch is repository independent, allowing reputation to be shared between projects. This may be especially useful for projects within organisations which require teams to work on multiple projects at a time, or for projects which are so large they are spread across multiple repositories.

8.1.4 Advanced User Permissions

For the purposes of this study, the Legit specification has defined a skeleton framework by which reputation can be used to determine basic user permissions. For uses of Legit in complex projects, it may be necessary to refine these, possibly allowing more specific permissions, such file-level access permissions within locked branches.

There may be other external factors which a group may wish to have contribute towards reputation, which Legit on its own would have no way to measure. In order to allow the inclusion of these factors, future work would need to determine a safe way to include external factors into the system, without compromising Legit's ability to validate itself.

8.2 Final Thoughts

This project has analysed the workflows and tools currently available to developers to aid in the process of creating software, and has attempted a novel take on this. A. Kelly states "*Software development, in all its forms, is an exercise in learning. Learning occurs within the teams that develop the software – not just amongst the managers*" [26] p.1 – it is clear that in order to empower developers to produce excellent software, we must ensure they fit comfortably into a well organised team.

Legit's decentralised take on peer approval facilitates this, by taking authoritarian trust away from the static central points in traditional corporate and open source systems, and spreading it to the team that develops the software via quantifiable, verifiable and agreeable reputation. Within a Legit system, everything is public, auditable and, perhaps most importantly, standardised. Legit opens a project entirely to a team, empowering corporate groups to bond and work efficiently, and expanding the reach of open source projects, by making them open not only in practice, but also in culture.

Legit may not be the ultimate system to implement the perfect decentralised peer approval workflow, but it has successfully shown not only the drawbacks of traditional informal communication, but also a way to combat this via quantifiable reputation.

References[7]

[1] J. D. Herbsleb and A. Mockus, “An empirical study of speed and communication in globally distributed software development,” *IEEE Transactions on Sofrware Engineering,* vol. 29, no. 6, p. Abstract, 2003.

[2] W. S. Humphrey, T. A. Chick, W. Nicholas and M. Pomeroy-Huff, “Team Software Process (TSP) Body of Knowledge (BOK),” Carnegie Mellon University, 2010.

[3] A. Bieniusa, P. Thiemann and S. Wehr, “The Relation of Version Control to Concurrent Programming,” in *International Conference on Computer Science and Software Engineering*, Wuhan Hubei China, 2008.

[4] Jst, “bzr/hg/git performance,” Mozilla, 30 November 2006. [Online]. Available: https://web.archive.org/web/20100529094107/http://weblogs.mozillazine.org/jst/archives/2006/11/vcs_performance.html.

[5] Git, “A Short History of Git,” [Online]. Available: https://git-scm.herokuapp.com/book/en/v2/Getting-Started-A-Short-History-of-Git. [Accessed 24 April 2015].

[6] J. Barr, “BitKeeper and Linux: The end of the road?,” The Linux Project, 11 April 2005. [Online]. Available: http://archive09.linux.com/feature/44147. [Accessed 24 April 2015].

[7] R. McMillan, “After controversy, Torvalds begins work on "git",” PC World, 20 April 2005. [Online]. Available: http://www.pcworld.idg.com.au/article/129776/after_controversy_torvalds_begins_work_git_/. [Accessed 24 April 2015].

[7] Please note: IEEE referencing has been used over Harvard, as it is better suited to technical references. Harvard has been used for the Bibliography.

[8] Eclipse Foundation, “Eclipse Community Survey 2014 Results,” 2014. [Online]. Available: https://ianskerrett.wordpress.com/2014/06/23/eclipse-community-survey-2014-results/.

[9] “Distributed Git - Distributed Workflows,” [Online]. Available: https://git-scm.herokuapp.com/book/en/v2/Distributed-Git-Distributed-Workflows.

[10] D. E. Perry, N. A. Staudenmayer, S. S. o. M. MIT and L. G. Votta, “People, Organizations, and Process Improvement,” *IEEE Software,* vol. 11, no. 4, pp. 36-45, 1994.

[11] Google, “Gerrit Documentation,” [Online]. Available: https://gerrit-documentation.googlecode.com/svn/Documentation/2.5.2/index.html.

[12] GitHub, “Features of GitHub,” [Online]. Available: https://github.com/features. [Accessed 24 April 2015].

[13] Kallithea, “Kallithea SCM,” [Online]. Available: https://kallithea-scm.org/#features. [Accessed 24 April 2015].

[14] Review Board, “Review Board,” [Online]. Available: https://www.reviewboard.org/. [Accessed 24 April 2015].

[15] RhodeCode, Inc, “Features of RhodeCode,” [Online]. Available: https://rhodecode.com/features/. [Accessed 24 April 2015].

[16] G. v. Rossum, “An Open Source App: Rietveld Code Review Tool,” Google, Inc, May 2008. [Online]. Available: https://cloud.google.com/appengine/articles/rietveld. [Accessed 24 April 2015].

[17] Atlassian, “Collaborative peer code review,” [Online]. Available: https://www.atlassian.com/software/crucible/overview/feature-overview. [Accessed 24 April 2015].

[18] Atlassian, “The best way to use Git with JIRA,” [Online]. Available: https://www.atlassian.com/software/stash. [Accessed 24 April 2015].

[19] Apache Software Foundation, “Return of Organization Exempt From Income Tax,” 30 April 2013. [Online]. Available: https://www.apache.org/foundation/records/990-2012.pdf. [Accessed 20 April 2015].

[20] Creative Services Ltd, “Establishing a Good eBay Reputation - Buyer or Seller,” Ebay, [Online]. Available: http://www.ebay.co.uk/gds/Establishing-a-good-eBay-reputation-Buyer-or-Seller-/10000000001338209/g.html.

[21] Ebay Inc, “Feedback scores, stars, and your reputation,” [Online]. Available: http://pages.ebay.co.uk/help/feedback/scores-reputation.html.

[22] Stack Exchange, “What is Reputation? How do I earn (and lose) it?,” [Online]. Available: http://stackoverflow.com/help/whats-reputation.

[23] Wikipedia, “Requests For Adminship,” [Online]. Available: https://en.wikipedia.org/wiki/Wikipedia:Requests_for_adminship.

[24] J. Atwood, “Stack Exchange Moderator Elections Begin,” Stack Exchange, 10 February 2012. [Online]. Available: http://blog.stackoverflow.com/2010/12/stack-exchange-moderator-elections-begin/. [Accessed 20 April 2015].

[25] Test Anything Protocol Team, “Test Anything Protocol,” [Online]. Available: https://testanything.org/. [Accessed 24 April 2015].

[26] A. Kelly, Changing Software Development: Learning to Become Agile, Chichester: John Wiley & Sons Ltd, 2008.

Bibliography

Collins-Sussman, C., Fitzpatrick, B. W., Pialto, C. M., 2004. *Version Control with Subversion*. Sebastopol: O'Reilly Media.

Jones, C., 2009. *Software Engineering Best Practices*. New York: McGraw-Hill.

Kelly, A., 2012. *Business Patterns for Software Developers*. Chichester: John Wiley & Sons Ltd.

McHale, J., Chick, T. A., Miluk, E., 2010. *Implementation Guidance for the Accelerated Improvement Method (AIM).* Pittsburgh: Carnegie Mellon University.

Watts, H., 1999. *Introduction to the Team Software Process*. Boston: Addison Wesley.

Appendix A

Original Project Brief

Context

Code collaboration normally follows an executive model, in which a person, or group of people, have full permissions to commit changes to the project. Users outside this group are normally not allowed to make edits, or they are able to suggest edits which members of the executive group can approve or reject.

MediaWiki and Stack Exchange use peer review systems to collaboratively achieve their goals, either to create content or to answer questions, and have a concept of earning trust in order to gain additional editorial or administrative privileges, although the exact procedure for gaining "trust" differs between them.

Aims

This project will investigate the possibilities of a peer-collaboration/approval model for the purposes of code development, predominantly on open source projects. Phase one of the project will compare the "trust" schemes used by MediaWiki and Stack Exchange, with the aim of creating a trust-based privileges model which code be edited in conjunction with git. An important goal of this will be to determine how best to set initial & ongoing privileges in a structure which best balances convenience with suitable security.

The second phase of the project will look at implementing the model as an extension to git, such that it can be used in a distributed system. The main aim for this stage is to investigate how best to store and authenticate a user's "reputation" such that each git repository knows if a commit has meets the appropriate criteria of privileges / approvals to be accepted. Specifically, the project will compare the benefits of a totally distributed system, in which each peers might sign to witness increases in reputation vs centralised authorities, which keep track of all the contributors and their privileges.

Appendix B

Technical Specification

File Structure

A folder named `.tracking` should be placed in the root of the directory, within an orphan branch, named `tracking`. The structure is initiated as follows:

```
.tracking/
   |- proposals/
   |   |- open
   |   |- pending
   |- users/
       |- (empty)
```

File Format

All files within this directory shall, unless otherwise stated, consist of any number of name value pair headers, followed by body text:

```
file   := *(header) "\r\n" body
header := name ":" value "\r\n"
name   := <Any letter, number or hyphen, leading and trailing whitespace
is ignored>
value   := <Any (US-ASCII) CHAR, other than "\r" or "\n", leading and
trailing whitespace is ignored>
body   := <Any (US-ASCII) CHAR)>
```

Adding a User to the System

Permission to Add a User

In order to add a new user to the system, a user must have at least the reputation points given by`git config --score.toAddUser`.

User File Structure

Each user has a *USER FILE* in the `.tracking/proposals/` directory. The name of this file shall be the email address of the user (as defined by RFC2822), with "@" replaced with "_". This file should be initialised as follows:

```
User: <Name of the user>
Proposals: 0
Accepted: 0
```

```
Rejected: 0
Reviews: 0
Bad-Rejects: 0
Bad-Accepts: 0
Good-Accepts: 0
Good-Rejects: 0
```

Adding a New User

When adding a new user, a peer should:

- Create a USER FILE for that user in `.tracking/users/`

This change should be in a single commit on the `tracking` branch, with the message:
`Added User: <name> <email>`

Proposals

By default, standard contributors cannot commit directly to a locked branch, instead they must submit a proposal branch to be considered and voted on.

Permission to Propose

In order to make a proposal, a user must have at least the reputation points given by`git config --score.toPropose`.

Proposal Structure

Code

A *PROPOSAL* is a branch of a LOCKED BRANCH containing one or more consecutive commits, referenced with `refs/heads/proposals/<id>`, where `<id>` is the commit hash at the head of the PROPOSAL.
A PROPOSAL is said to be "**OF** branch `<x>`" if it is based on either:

- A commit in the branch `<x>`
- A commit in another PROPOSAL, which is itself of branch `<x>`
    - *Note: It does not need to be the head commit*
- A proposal must be OF a LOCKED BRANCH.

Meta Data

The meta data for a PROPOSAL shall be stored in the tracking branch, under:`.tracking/proposals/<id>/`, where `<id>` is the commit hash at the head of the PROPOSAL. This directory is called the *PROPOSAL DIRECTORY*.

File names

There should always be one file within the PROPOSAL DIRECTORY named "proposal". Other file names are used for REVIEWS and are named after the reviewer:

```
filename := "proposal" / review
review   := <Addr-Spec as defined by RFC2822, with "@" replaced with "_">
```

Proposal File

When creating a PROPOSAL, the PROPOSER should create a file named "proposal" in the PROPOSAL DIRECTORY.

Headers

Header	Description
Proposer	The name and email address of the proposer, as specified byRFC2822 *(required)*
Votes	The numerical score of the number of NET VOTES *(required)*
Status	The status of the review. One of: Open Accepted Rejected *(required)*
Submitted-at	The RFC2822 timestamp at the time of submmiting the proposal*(required)*
Extension-of	Specifies the id of the PROPOSAL this extends *(required when this proposal is an extension)*
Extended-by	This header may appear multiple times. It specifies the ID of a PROPOSAL which extends this proposal *(required when extension proposals for this proposal exist)*
Fix-of	Specifies the id of the PROPOSAL this fixes *(required when this proposal is a fix)*
Alternative	This header may appear multiple times. It specifies the ID of a PROPOSAL which fixes this proposal *(required when fix proposals for this proposal exist)*
Merged-By	This header may appear multiple times. It specifies the ID of a PROPOSAL which merges this proposal into the LOCKED BRANCH *(required when merge proposals for this proposal exist)*
Merge-of	Specifies the ID of a PROPOSAL which this PROPOSAL merges into the LOCKED BRANCH *(required when this proposal is a merge proposal)*

Submitting a Proposal

When submitting a PROPOSAL, a PROPOSER should:

- Create the PROPOSAL DIRECTORY and fill the "proposal" file with their details, and the details of the proposal, for example:

```
Proposer: Joe Bloggs <joe.blogs@example.net>
Submitted-at: Mon, 22 Feb 2015, 16:59:00 +0000
Votes: 0
Status: Open
[Extension-of: ID]
[Merge-of: ID]
[Fix-of: ID]

Text describing the proposal...
```

- Update their USER FILE, incrementing the value of the `Proposals` Header by one.
- Update the "open" file in `.tracking/proposals/`, appending the `<id>` to a new line at the bottom.

These changes should be in a single commit on the `tracking` branch, with the message:
`Submitted Proposal: <id>`

Types of Proposals

All *NORMAL* PROPOSALS are based directly on the LOCKED BRANCH, however when they are based instead on another PROPOSAL, they must be one of the following types:

Extensions

An **EXTENSION** is a PROPOSAL which is based on another PROPOSAL, and relies on features that it introduces. When an EXTENSION is APPROVED, it is *not* merged until all PROPOSALS below it are APPROVED.

As such, when a PROPOSAL is APPROVED, the client should also check if there are any APPROVED EXTENSIONS and MERGE them at the same time.

Fixes and Alternatives

A **FIX** is a PROPOSAL which modifies features that another PROPOSAL introduces. A FIX does not need to be based on the PROPOSAL which it fixes.

Approving a Fix

When a FIX is APPROVED, any other FIXES for the same PROPOSAL are automatically deemed to be REJECTED.

When the FIX is based on the PROPOSAL it fixes

A PROPOSAL on which a FIX is based does *not* need to be APPROVED in order for an APPROVED FIX to be MERGED.

When the FIX is not based on the PROPOSAL it fixes

A PROPOSAL on which a FIX is not based is automatically deemed to be REJECTED when the FIX is APPROVED.

Merges

When MERGING an APPROVED PROPOSAL results in a merge conflict, any user may attempt to solve the merge, and submit it as a new *MERGE PROPOSAL*. A MERGE PROPOSAL must be a single commit, with the parents being:

- The head of the original PROPOSAL
- The head of the LOCKED BRANCH OF the original PROPOSAL

When a MERGE PROPOSAL is APPROVED, any other MERGE PROPOSALS for the same PROPOSAL are automatically deemed to be REJECTED.

Reviews

Permission to Review

In order to make a *REVIEW*, a user must have at least the reputation points given by`git config --score.toReview`.

Review Structure

REVIEWS are saved in the PROPOSAL DIRECTORY in a file named after the REVIEWER (See `Proposal Structure` > `Meta Data` > `File Names`).

Headers

Header	Description
Reviewer	The name and email address of the REVIEWER, as specified byRFC2822 *(required)*
Reviewed-at	The RFC2822 timestamp at the time of submmiting the REVIEW*(required)*
Result	Either "Accept" or "Reject"

Submitting a Review

When reviewing a PROPOSAL, a REVIEWER should:

- Create a file in the PROPOSAL DIRECTORY. The filename should be their email address, with "@" replaced with "_", (See `Proposals` > `Structure` > `Meta Data` > `File Names`) and fill it with their review information, for example:

```
Review: John Doe <john.doe@example.net>
Submitted-at: Mon, 22 Feb 2015, 16:59:00 +0000
Result: Accept

Text describing the review...
```

- Update their USER FILE, incrementing the value of the `Reviews` Header by one.
- Update the "proposal" file, with an incremented value for the `Votes` Header:

```
incremented_vote := old_vote_header_value + reviewer_vote

reviewer_vote    :=  1 if decision = "Accept"
                 := -1 otherwise
```

These changes should be in a single commit on the `tracking` branch, with the message:
`Reviewed Proposal: <id>`

Accepting or Rejecting a Proposal

If, after reviewing a proposal, the proposal has reached either the positive or negative number of required votes, the peer should close the PROPOSAL:

Rejecting a Proposal

A peer rejecting a proposal MUST make the following changes to the `tracking` branch in a single commit, with the message `Rejected Proposal: <id>`:

- For each reviewer, update their USER FILE:
  - Increasing the number of `Bad-Rejects` if the reviewer voted to accept the PROPOSAL
  - Increasing the number of `Good-Rejects` if the reviewer voted to reject the PROPOSAL
- Update the original PROPOSER'S USER FILE, increasing the number of `Bad-Proposals` by one.
- Update the "proposal" file, changing the status to "Rejected".
- Remove the PROPOSAL's `<id>` from the "open" file in `.tracking/proposals/`.
- Update the "pending" file in `.tracking/proposals/`, appending the `<id>` to a new line at the bottom.

Accepting a Proposal

Marking as Accepted

A peer marking a proposal as ACCEPTED MUST make the following changes to the `tracking` branch in a single commit, with the message `Accepted Proposal: <id>`:

- For each reviewer, update their USER FILE:
  - Increasing the number of `Good-Accepts` if the reviewer voted to accept the PROPOSAL
  - Increasing the number of `Bad-Accepts` if the reviewer voted to reject the PROPOSAL
- Update the original PROPOSER'S USER FILE, increasing the number of `Good-Proposals` by one.
- Update the "proposal" file, changing the status to "Accepted"

Merging

A peer ACCEPTING a PROPOSAL MUST check to see if the PROPOSAL can be merged into the LOCKED BRANCH. An *APPROVED* PROPOSAL may be merged if:

- It is based directly on the LOCKED BRANCH, or
- It is based on a PROPOSAL which has been merged, or
- It is based directly on a PROPOSAL for which this is a FIX, and that PROPOSAL meets one of these three conditions.

If the PROPOSAL matches one of the above conditions, the peer MUST attempt to merge the PROPOSAL into the LOCKED BRANCH.

Merging without Conflict

If the PROPOSAL can be merged into the LOCKED BRANCH without causing a merge conflict, the peer MUST merge it into the LOCKED BRANCH in a single commit with the message `Merged Proposal: <id>`. This commit may *only* be a merge - no new lines of code may be added.

The peer should then remove the PROPOSAL's `<id>` from the "pending" file in `.tracking/proposals/`, commiting that to the `tracking` branch with the commit message: `Merged Proposal: <id>`.

In the case of a Merge Conflict

If merging the PROPOSAL into the LOCKED BRANCH causes a merge conflict, the peer MAY attempt to create a MERGE PROPOSAL.

When submitting the MERGE PROPOSAL to the `tracking` branch, the peer MUST add the `Merged-By` header to the PROPOSAL it is based on.

When a MERGE PROPOSAL is accepted and successfully merged into a LOCKED BRANCH, the merging peer should then remove the id of both the MERGE PROPOSAL and the original PROPOSAL from the "pending" file in `.tracking/proposals/`, committing this to the `tracking` branch with the commit message: `Merged Proposal: <id of original PROPOSAL> via <id of MERGE PROPOSAL>`.

Appendix C

Source Code

Legit's source code, along with the code from Git on which Legit is based, were handed in along with this project. Noteworthy files that were created or amended by this project are as follows:

- git-legitimize.sh[8]
- git-legitimise.sh
- git-legit-setup.sh
- git-add-contributor.sh
- git-propose.sh
- git-proposal-details.sh
- git-review.sh
- git-check-commit.sh
- git-merge-proposal.sh
- git.c[9]

Test files are found in the t/ directory.

Install Notes

- If installing on Windows, a copy of mysysgit is recommended.
- If you already have git on your system, ensure you set $GIT_EXEC_PATH to the directory of Legit's source code before attempting to run any code.

[8] The US English spelling of "Legitimise" was used by default, to conform with Git's pre-existing standards. The UK Spelling is available as an alias.

[9] This file is largely unmodified, aside from some code to reroute "git commit" commands to Legit's "check-commit.sh" script.

Appendix D

Gantt Chart

Overleaf shows two sets of Gantt charts. The first page shows the original plan; the web site development was removed from the project due to risk mitigations and as the research showed this would not be additive to the overall project. This is shown in the actual progress, displayed by the second chart.

	February																							March															
	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Initial Stage																																							
Complete Project Plan	🟦																																						
Review Project Plan				🟦																																			
Review of existing tools		🟩	🟩	🟩	🟩	🟩																																	
Refine Functional Spec					🟩	🟩	🟩	🟩	🟩	🟩	🟩																												
Shell Implementation																																							
.tracking structure										🟧	🟧	🟧	🟧	🟧	🟧																								
Create new proposed commit																🟧	🟧	🟧																					
Display proposed commits																			🟧	🟧	🟧																		
Permissions Calculation																						🟧	🟧	🟧	🟧														
Accept a proposed commit																										🟧	🟧	🟧	🟧										
Validation Algorithm																														🟧	🟧	🟧	🟧						
Signatures																																		🟧	🟧	🟧			
Milestones																																					🟧	🟧	🟧
Testing															🟨			🟨			🟨			🟨	🟨			🟨	🟨			🟨	🟨			🟨			🟨

	March															April																											
	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Web Development																																											
.tracking structure	🟧	🟧	🟧																																								
Read proposed commits				🟧	🟧	🟧	🟧																																				
GUI for commits								🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧																												
Accept a proposed commit																🟧	🟧	🟧																									
List of developers and rankings																			🟧	🟧	🟧	🟧	🟧																				
Signatures																								🟧	🟧	🟧																	
Testing			🟨			🟨	🟨			🟨			🟨	🟨				🟨				🟨				🟨																	
Report																																											
Writing Report																											🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩								
Review Report With Supervisor																																			🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	
Handin																																											🟥

Reasearch																													
Peer Approval Models	🟦	🟦	🟦																										
Market Share of Git		🟦																											
Credibility, Reputation and Privilages						🟦	🟦	🟦																					
Privilages within Peer Approval Systems							🟦	🟦																					
Specification																													
Functional																													
Peer Approval and Voting			🟥																										
Reputation								🟥																					
Technical																													
Meta-Data Storage					🟥																								
Reputation										🟥	🟥																		
Working ontop of a Git repository										🟥	🟥				🟥			🟥											
Development																													
Manual Testing				🟦	🟦																								
Prototyping									🟦	🟦	🟦																		
Coding																													
Proposals												🟦	🟦		🟦		🟦		🟦										
Reviews														🟦	🟦		🟦		🟦										
Accepting Proposals																🟦	🟦												
Reputation																	🟦	🟦	🟦										
Testing													🟨		🟨		🟨	🟨	🟨										
Report																													
Git	🟥	🟥																											
Peer Approval Methods			🟥	🟥																	🟥								
Reputation							🟥	🟥																🟥					
Design Discussion											🟥			🟥				🟥		🟥	🟥	🟥							
Project Management and Testing Strat																					🟥	🟥	🟥						
Critical Evaluation																								🟥	🟥				
Conclusion																									🟥	🟥			
Graphics																													
Hand Drawn				🟥							🟥				🟥					🟥									
Neatly Drawn																🟥	🟥					🟥	🟥	🟥		🟥	🟥		
Editing and Refinement																🟥							🟥				🟥	🟥	🟥

Appendix E

Risk Assessment

The project's risk assessment is shown opposite, with a red/amber/green traffic light scheme supporting the impact scores for ease of reference. The top risk did materialise for this project, however the mitigation proved satisfactory in resolving this, to enable full completion of the project and report to standard and on time.

Risk	Description	Likelihood	Impact	Priority	Mitigation
Developer Health	The possibility of depression incapacitating developer.	0.9	1.0	0.9	Ensure project contains multiple milestones at which a working solution could be submitted. Use Bash rather than C as the primary language, to facilitate speedy development.
Project not technically viable	Decentralised peer approval is a new concept. Possibility this is not viable.	0.5	0.8	0.4	Prove viability early on in project via modelling. Utilise prototyping while defining technical specification to identify technical impossibilities before going into full development.
No viable solution to reputation	Reputation is not used in current code review software. Possibility it is not viable.	0.4	0.7	0.28	Research reputation within other systems before prototyping. In the event it is not possible to use in a code review context, reassess project goals.
Loss of work	Technical or physical issues may lead to loss of data.	0.3	0.9	0.27	All code kept under Git version control, and backed up to GitHub. All accompanying documents and files backed up via Microsoft OneDrive.
Difficulty working on top of Git	Git's source code may be difficult to work with.	0.3	0.4	0.12	Design functional specification such that Legit can be written as an application on top of Git, rather than an extension of it.