A framework for the best practise in Developing Java Web Applications with Version Control

Simbarashe Makwangudze

R09985v

2018

Submitted in partial fulfilment of the requirement for the degree of

Master of Science in Information Systems Management

Department of Computer Science and Information Systems in the

Faculty of Science and Technology at

Midlands State University

Gweru

May 2018

Supervisor: Mr. M. Zhou

Co-Supervisor: Mr. S. Furusa

ABSTRACT

Version Control Systems for quite some time present an integral part of development process and a must have tool for both individual developers and teams as well. However, use of version control systems and choice of proper workflow can at first be challenging. Developers and teams often do not invest enough time to get to know the possibilities of such systems, which results in these systems not being used to their full potential. The purpose of the study was to develop a framework for version control in the development of java web applications. Using a qualitative research in conjunction with design science paradigm, data was collected through semi-structured interviews, participant observation, document review and qualitative questionnaire. The analysis of data was done using structural coding to find emerging themes. The findings of the study revealed that version control was not implemented correctly and, in some situations, never used at all. This made it difficult to properly manage software releases, development teams could not effectively work together on a project and source code management needed more effort in order to combine code from different developers. In this research, I have mitigated this problem by identifying and systematizing useful tools and best practices in using version control systems. I have covered the case of Git – one of today's most popular version control system. This study recommends that the development team should use a framework for version control to allow software developers to collaborate, developers to properly manage source code, this will in turn increase productivity and software may then be released quicker with fewer bugs.

DECLARATION

I, Simbarashe Makwangudze, do hereby declare that the work in this dissertation titled, "**A framework for the best practise in Developing Java Web Applications with Version Control"**, is my original work and I have fully acknowledged all the sources used in this study by means of complete references.

…….. ……

Student's Signature (Simbarashe Makwangudze) Date

…….. ……

Supervisor's Signature (Mr Zhou) Date

…….. ……

Supervisor's Signature (Mr Furusa) Date

APPROVAL

This dissertation entitled **"A framework for the best practise in Developing Java Web Applications with Version Control"** by **Simbarashe Makwangudze** meets the regulation governing the award of the degree of **MSc. Information Systems Management** of the Midlands State University and is approved for its contribution to knowledge base and theory for future research.

ACKNOWLEDGEMENTS

Although this dissertation was coupled with long march and endurance, various people and organisations made it a success. Therefore, let me thank them for aiding me with the needed knowledge and support to overcome this educational hurdle.

Firstly, I would like to thank the Almighty God for the gift of life and strength to perform this work. That is the natural ability and the sprightliness of the energy to act accordingly; all the glory to God.

My deepest appreciation goes to my supervisors Mr Furusa and Mr Zhou for their persistence and convincing spirit of adventure in regard to academic research. They helped me with the necessary skills required in navigating the academic territory. Without their guidance and continual assistance this dissertation would not have been a success.

I would also like to sincerely thank the management, employees for Multipay. You exhibited great cooperation during the entire period of the research.

Lastly, let me thank the Midlands State University, in particular the department of Computer Science and Information Systems for affording me another opportunity to grow in academic respect and to face the challenges of coming up with this work.

TABLE OF CONTENTS

LIST OF APPENDICIES

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE: INTRODUCTION

1.1 INTRODUCTION

This chapter starts with an overview of version control systems in order to give the reader an insight on the study. This is then followed by the background study, which leads to the formulation of the problem statement. It also presents the research objectives, questions, significance of the study, the motivation and research gap as well as delimitation of the study.

1.1.1 OVERVIEW OF VERSION CONTROL SYSTEMS

By developing programs, software engineers produce source code. They change and extend it, undo changes, and jump back to older versions. When several software engineers want to access the same file, concurrency becomes an issue. Version Control Systems (VCSs) enable the acceleration and simplification of the software development process and enable new workflows. They keep track of files and their history and have a model for concurrent access. Keeping track of changes has become an essential part of the practice of software development. The modern complexities of the craft are such that programmers must manage source code not only spatially, in terms of files and directory structures, but also temporally, over complex, diverging and converging timelines resulting from multiple strands of work done in parallel. A class of tools called version control systems (VCSs) has been developed to help address these temporal concerns, allowing programmers to more easily navigate through and communicate about the changes they make to source code. The growing sophistication of these tools has had a transformative impact on the practice of programming (Rigby 2013; Spindles 2012) and has in turn prompted an explosion in the number and diversity of collaboratively developed software projects (Doll 2013).The motivation for usage of VCS is to amongst other things allowing collaboration amongst different stakeholders in a software project, coordination of tasks ,store versions of the software and track changes of configuration files, source code and developer activity as a software project progresses.

1.2 BACKGROUND OF THE STUDY

Despite considerable progress, version control systems continue to bump up against certain stubborn difficulties. For users of these systems, manually reconciling conflicting change a frequent occurrence when multiple programmers work on the same piece of code remains a frustrating and error prone process. For the systems' designers, the choice of appropriate algorithms and representations for keeping track of versions for example, the choice of

whether to record history as a series of transformation or as snapshots remains a challenge and a point of dispute.

Meanwhile, newly opened avenues for collaboration in software development are putting pressure on previously stable concepts and practices. By leveraging features of newer version control systems, web-based services like GitHub, which host the source code, wikis, and issue trackers of a growing number of open source projects*, have made it trivial for anyone to create clones of those projects. These copies, known as "forks", allow a user to split off and continue development of the project independently of the original. Some of the changes introduced in these clones eventually find their way back into the project they split from, and some diverge off in their own direction. This can lead to tangled histories and a proliferation of subtly different variants, that is, projects whose source code is licensed and published by the copyright holder with provisions that allow for open modification and/or repurposing. Weber (2004) for an overview of open source software practices.

The research presented here is an attempt to bring to the surface some of the issues faced by users and designers of modern version control systems. The core of the project is an analysis of several popular VCSs (namely Git, Subversion, and Mercurial), a comparison of their differing approaches to dealing with change, and ultimately an explication of their limitations and recommending the best practice to implement version control. The argument is that many of those shortcomings can be traced down to a number of long standing conceptual problems issues of identity, representation, reference, and meaning not just in version control but in computing and information practice in general.

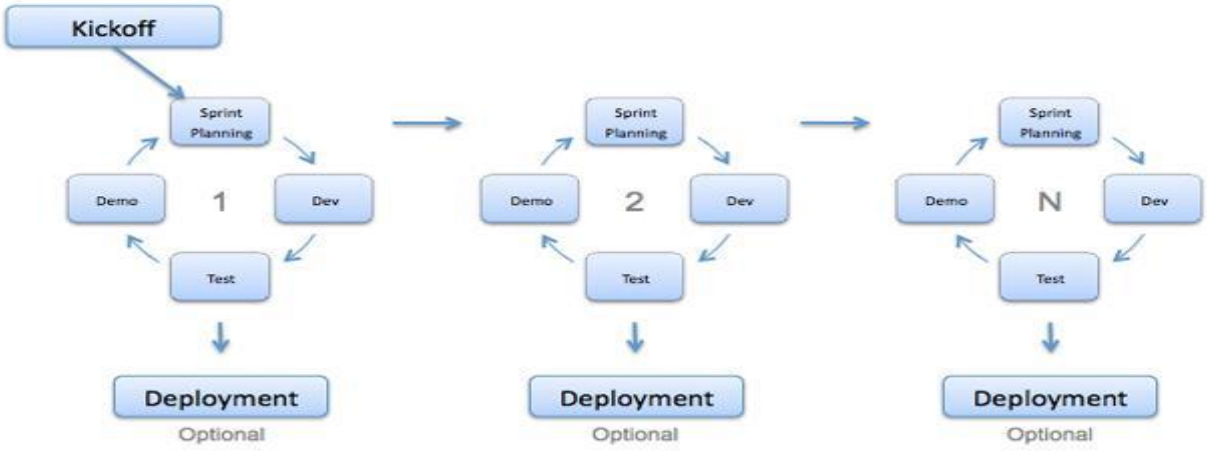

Figure 1.1 Agile development model

Source: https://www.tutorialspoint.com/sdlc/sdlc_agile_model.htm

As teams² design, develop and deploy software, it is common for multiple versions of the same software to be deployed in different sites and for the software's developers to be working simultaneously on updates. Bugs or features of the software are often only present in certain versions (because of the fixing of some problems and the introduction of others as the program develops). Therefore, for the purposes of locating and fixing bugs, it is vitally important to be able to retrieve and run different versions of the software to determine in which version(s) the problem occurs. It may also be necessary to develop two versions of the software concurrently: for instance, where one version has bugs fixed, but no new features, while the other version is where new features are worked on. This scenario is shown below (Driessen 2010).

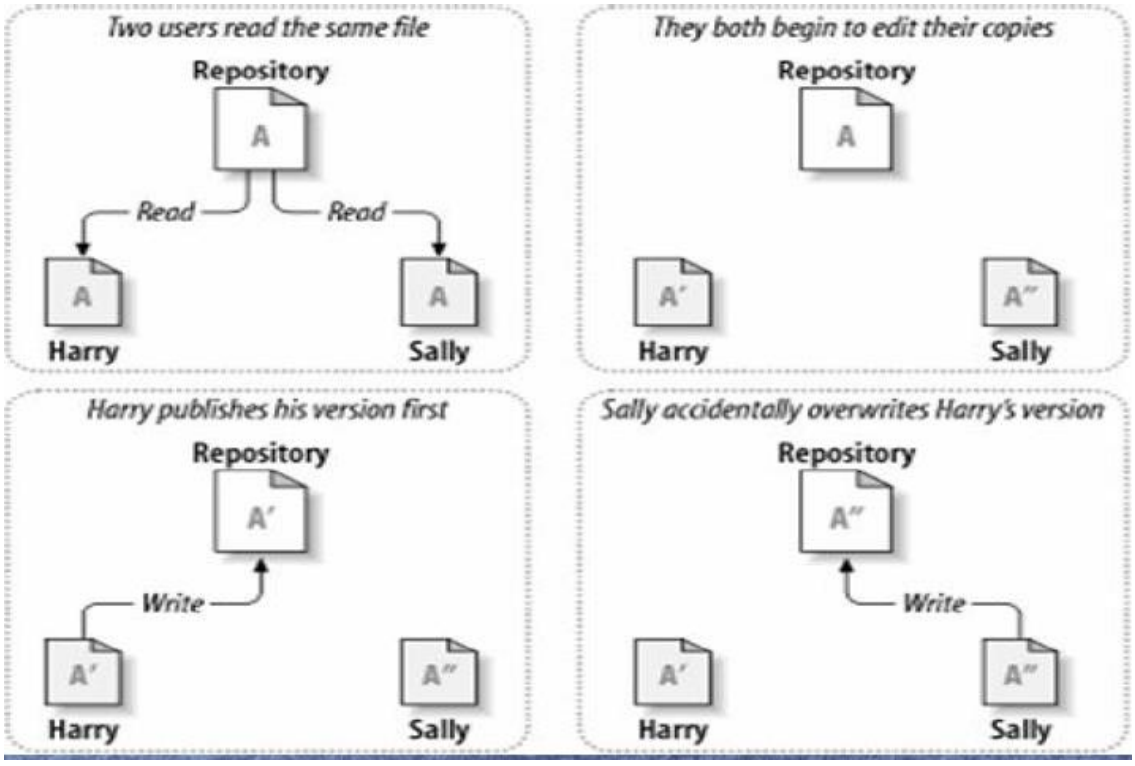

Figure 1.2 Without Version Control

At the simplest level, developers could simply retain multiple copies of the different versions of the program, and label them appropriately. This simple approach has been used in many large software projects. (Waleed *Mohamed Mahmoud Al-Adrousy,2012*). While this method can work, it is inefficient as many near-identical copies of the program have to be maintained. This requires a lot of self-discipline on the part of developers and often leads to mistakes. Since the code base is the same, it also requires granting read-write-execute permission to a set of developers, and this adds the pressure of someone managing permissions so that the

code base is not compromised, which adds more complexity. Consequently, systems to automate some or all of the revision control process have been developed. This ensures that the majority of management of version control steps is hidden behind the scenes.

Moreover, in software development, it has become increasingly common for a single document or snippet of code to be edited by a team, the members of which may be geographically dispersed and may pursue different and even contrary interests. Sophisticated revision control that tracks and accounts for ownership of changes to documents and code may be extremely helpful or even indispensable in such situations.

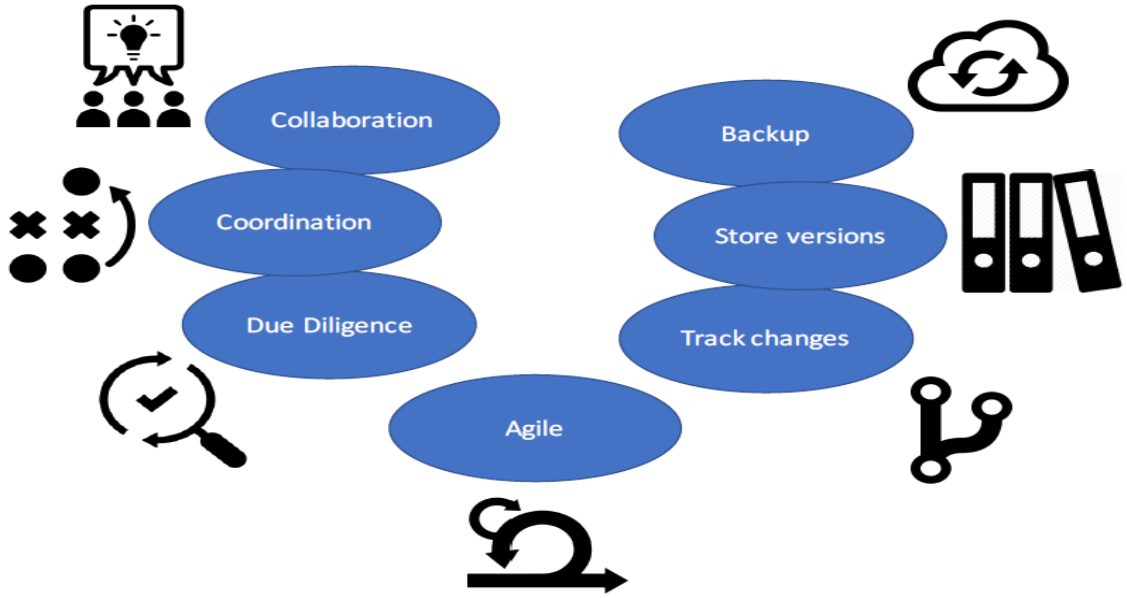

Figure 1.3: Motivation for Version Control

Hence it is against this background that the researcher, saw he need to introduce a framework for the best practice in using version control when developing software projects particularly in this case java web applications. As shown in the diagram VCS allow collaboration, coordination, implicitly backup and software developers are able to store different versions and track changes.

1.3 STATEMENT OF THE PROBLEM

In the background, issues that arose as a result of not having a framework for implementing version control in modern enterprise Java web applications were noted and as a result of the above analysis it is worthy to mention that for single developers, it's already a challenge to keep track of different versions of a program, not to mention all the different backups the developer might take. Now if the developer is working as part of a team on a project that involves text, multimedia, and coding in short, a wide variety of information in the works and each data or program file is worked on by several people ("concurrency") each person adding, saving, editing, and saving again. How on will the technical lead and other developers know who did what, when, and why, and be able to track all of these changes?

It has also been noted that the current state of not implementing version control has brought about issues. Manual backup is human dependent, if a person forgets to backup an important file and it gets damaged, that is a step backwards and this can lead to a project failure. Without version control it is impossible to work in a team environment where multiple people are contributing towards the final solution – members can easily overwrite each other work without knowing. In scenarios in which hundreds of people are working on the same project, a lack of management and coordination can have drastic implications on the project's completion. By tracking and updating the changes every contributor makes to the 'big picture,' there is less room for error and more room for productivity and success. Every time someone goes to work on the project, they should know exactly what has already been completed, added, changed, and so on. Otherwise, they might end up working on something that's already been finished – or worse, something that has no applicability to the project anymore.

When a fault appears in the software due to an upgrade, it is difficult to revert back to the last working copy, unless manual backup is done at every step, which cannot be relied upon. If a fault appears in the system, there is no accountability, because there is no track as to who made that change, and it completely defeats any attempt at due diligence of the software source before release.

1.4 RESEARCH OBJECTIVES

The primary research objective is:

To develop a framework for the best practice in implementing version control in modern enterprise java applications.

In order to fulfil the above stated primary objective, 3 secondary objectives have been defined.

The secondary research objectives are:

- To examine the readiness of the development department to adopt version control in software development.
- To explore and suggest improvements on providing a secure way of accessing source code.
- To examine how the department can manage software release with version control.

1.5 RESEARCH QUESTIONS

This section defines the research questions of the study that was used to answer the objectives stated in the previous section.

The following main research question has been formulated from the main research objective:

HOW TO DEVELOP A framework FOR HAVING the best practices IN implementing version control in modern enterprise java web applications?

The study was guided by five research questions stated as follows:

- What is the best approach to effectively allow developers to collaborate.
- How will version control allow concurrency in projects
- How can software testers make use of software version control software when testing new software versions, approving software version, maintaining automated tests, creating release notes and code review prior to release.

1.7 RESEARCH METHODOLOGY

This research is confined to one company Multipay Solutions. Further, within this business, the research will concentrate on the software development department which is primarily involved in software development.

The systematic way to find out more about things resulting in knowledge increase is termed 'Research' according to Saunders et. al. (2009). This 'systematic way' of studying using available and relevant material and sources to establish facts and confirm or arrive at new conclusions suggests that research is based on rational associations of different realities according to Ghauri and Gronhaung (2005) as quoted by Suanders et al. (2009). Thus, 'Research methodology', according to Fisher (2010), is a framework within which a research is undertaken. He goes on to add that it forms the base from which research is initiated, provides a platform to get answers on what is being explored as well as a framework on the exploration process.

1.7.1 RESEARCH APPROACH

Two types of approaches are outlined here: the deductive and the inductive approach. The researcher will use a deductive approach.

Deductive Approach

It is also called the testing theory. The deductive approach develops the hypothesis or hypotheses upon a pre-existing theory and then formulates the research approach to test it (Silverman, 2013). This approach is best suited to contexts where the research project is concerned with examining whether the observed phenomena fit with expectation based upon previous research (Wiles et al., 2011). The deductive approach thus might be considered particularly suited to the positivist approach, which permits the formulation of hypotheses and the statistical testing of expected results to an accepted level of probability (Snieder & Larner, 2009). However, a deductive approach may also be used with qualitative research techniques, though in such cases the expectations formed by pre-existing research would be formulated differently than through hypothesis testing (Saunders et al., 2007). The deductive approach is characterised as the development from general to particular: the general theory and knowledge base is first established and the specific knowledge gained from the research process is then tested against it (Kothari, 2004).

1.7.2 RESEARCH STRATEGIES

The research strategy is how the researcher intends to carry out the work (Saunders et al., 2007). The strategy can include a number of different approaches, such as experimental research, action research, case study research, interviews, surveys, or a systematic literature review. The following research strategies will be used:

Action research

The researcher has chosen this research strategy as he will be actively involved in the processes. Action research is a form of applied research where the researcher attempts to develop results or a solution that is of practical value to the people with whom the research is working, and at the same time developing theoretical knowledge. Through direct intervention in problems, the researcher aims to create practical, often emancipatory, outcomes while also aiming to rein form existing theory in the domain studied. As with case studies, action research is usually restricted to a single organisation making it difficult to generalise findings, while different researchers may interpret events differently. The personal ethics of the researcher are critical, since the opportunity for direct researcher intervention is always present.

1.7.3 RESEARCH DESIGN

The researcher will make use of qualitative research design. Qualitative research design according to Harrell & Bradley (2009), is an approach aimed at understanding the experiences of human beings by applying specific research methods such as interviews, observations, qualitative questionnaires and document review. Qualitative research put emphasis on words in the collection and analysis of data in order to facilitate the generation of themes and patterns emerging from a study (Williams, 2007; Owen, 2014). The qualitative research approach was chosen to obtain views, opinions and experiences the current processes in version control systems. It also sought to understand the efforts made by the development team to implement version control at different stages of software development... The information was critical in assisting the researcher to build or develop the proposed framework.

1.8 DATA COLLECTION METHODS

Data collection stresses the development of logical research evidence that facilitate information gathering in a specific research (Ololube, Kpolovie & Harcourt, 2012). Data can be collected from both primary and secondary sources in order to provide evidence of a

researcher. Yin (2014) stated that there are six (6) different sources for evidences which researchers could use during data collection in a case study research. These include documentation, archival records, interviews, direct observations, participant observation, and physical artefacts. The following data collection methods were used during the study.

"A research method is a technique for (or a way of proceeding in) gathering evidence. One could reasonable argue that all-evidence gathering techniques fall into one of the following three categories: listening to (or interrogating) informants, observing behaviour, or examining historical traces and records. In this sense, there are only three methods of social inquiry".

LITERATURE REVIEW

Literature review is the study and acknowledgement of other scholars' researches that are in line with the current study. This research instrument is commonly used in collecting secondary data. The secondary data that were included in the study were obtained from journal articles and books. A literature review was carried out to enable the researcher to establish the route of the study. Models, frameworks and methodologies that were found useful were used in this study to understand the concepts and theories required in version control systems.

INTERVIEWS

The purpose of an interview as a research instrument is to dig into the people's views and experiences about certain issues to be studied (Gill et al., 2008). They are basically grouped into three basic categories: structured, semi-structured and unstructured. Semi-structured interviews which involve a series of open-ended questions based on the topic areas the researcher wants to cover (Harrell & Bradley 2009), were used in this study. This type of the interview enabled the researcher to encourage the participants to give more details on previous responses that could have been partially answered.

In designing the interview questions, the researcher followed a general interview guide approach protocol (Bricki & Green, 2007; Jacob & Furgerson, 2012) also known as interviewing the participant based on the themes. The interview questions were arranged under broad areas identified as research questions in the **Chapter One** in order to ask every participant the same question.

The interviews started with the software developers and concluded with the software developer's managers. This enabled the researcher to clarify some questions which the developers were not comfortable to answer.

The researcher was an active participant throughout the process. He was given access to source code and was actively involved in the set up of version control systems.

OBSERVATIONS

This is a method of intentionally noticing, transcribing and/ recording events that have been seen in the selected area of study (Runeson & Höst, 2009). According to Brinkerhoff, Ortega & Weitz (2013), observation is a way of conducting a research by participating, observing cases in the field and/or interviewing where necessary. The details of observation may be in the form of notes, images, videos or voice. Basically, there are two types of observations in data collection. These are participant and non-participant observations.

Participant observation is more appropriate in qualitative research since it gives the researcher the opportunity to collect data on a natural setting in places believed to be relevant to the research questions (Mack et al., 2011). This study used participant observation where the research took part in the phenomenon being studied (Palmer & Griggs, 2010; Gratton & Jones, 2010) by acting as an employee of the participating organization. The researcher observed the steps taken to setup version control systems, the usage and various commands needed to run the software.

The researcher also made notes and made video tutorials as the team were setting up the software. The major drawback of this method is that the researcher was required to establish himself as the employee of the participating organization in order to avoid awkward behavior. This type of data gathering technique also requires the skill of a researcher to balance the role of being an observer and an employee (Neale, 2009). Participant observations, as further stated by Neale (2009), put participants at risk, especially when studying prohibited/illegal activities.

DOCUMENT REVIEW

Document review is a way of collecting data by going through existing documents concerning the study to find relevant information that can be used for data analysis (CDC, 2009). This method can be used to triangulate other methods being used in the study to improve data validity. This method was used to suggest questions for further inquiry during data collection. Documents included manuals and screenshots from the software manuals.

The documents were sorted according to the major headings described in the interview protocol.

EXPERT REVIEW

After the development of the initial framework, expert reviewers were presented with the framework to analyse the features necessary to be portrayed in the framework. The findings from expert reviewers were used to refine the framework as prescribed by Hevner et al. (2004) that evaluation of the framework should be done before it can be applied to the appropriate environment. Their contributions/ comments are presented in Appendix C.

Data collection and analysis is dependent on the methodological approach used (Bryman, 2012). The process used at this stage of the research contributes significantly to the study overall reliability and validity (Saunders et al., 2007). Regardless of the approach used in the project, the type of data collected can be separated into two types: primary and secondary.

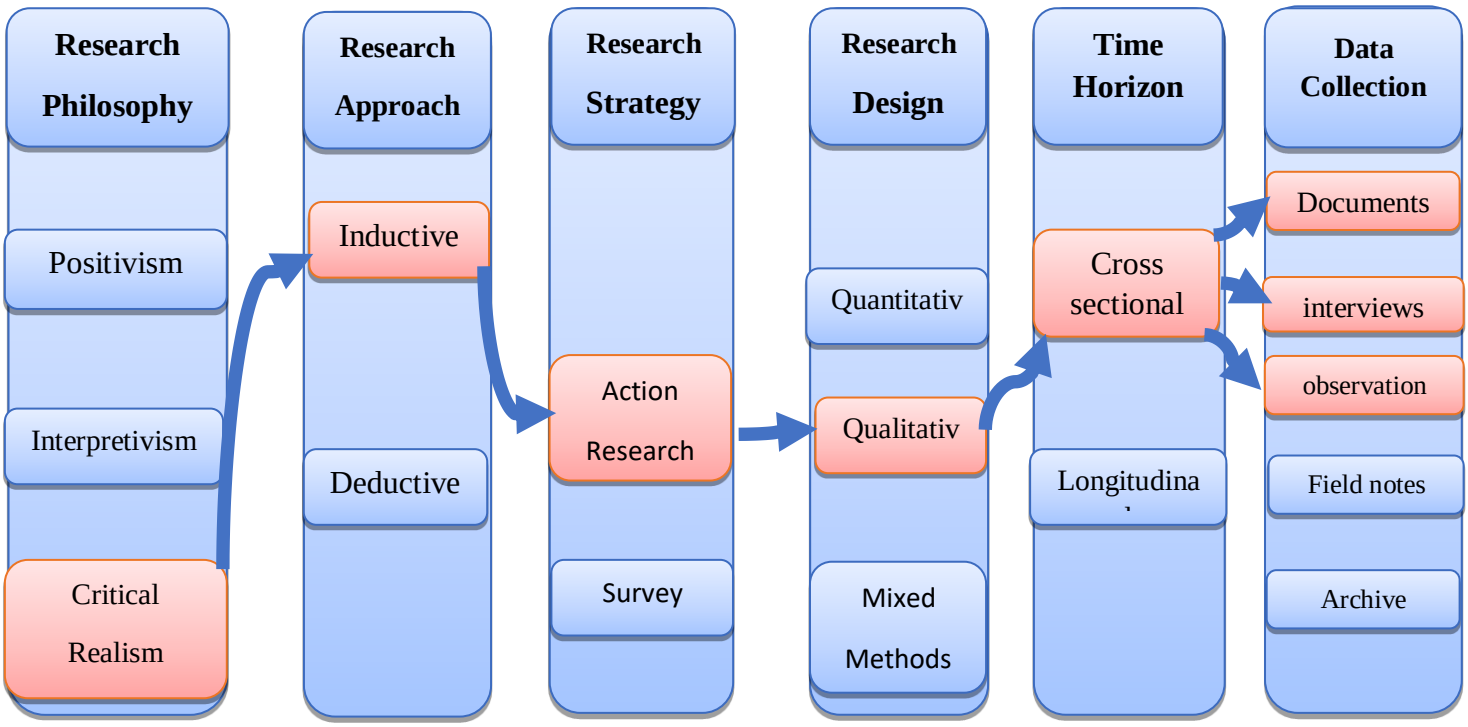

Figure 1.5 Summary of Research Methodology Adopted

Source: E-Government Implementation for inter-organisational Information sharing, Ephias Ruhode

1.8.1 DATA ANALYSIS

It is the process of systematically applying statistical and/or logical techniques to describe and illustrate, condense and recap, and evaluate data. According to Shamoo and Resnik (2003) various analytic procedures "provide a way of drawing inductive inferences from data and distinguishing the signal (the phenomenon of interest) from the noise (statistical fluctuations) present in the data".

The goals of the data analysis are:

- To make some type of sense out of each data collection
- To look for patterns and relationships both within a collection, and also across collections, and
- To make general discoveries about the phenomena you are researching.

While data analysis in qualitative research can include statistical procedures, many times analysis becomes an ongoing iterative process where data is continuously collected and analysed almost simultaneously. Indeed, researchers generally analyse for patterns in observations through the entire data collection phase (Savenye, Robinson, 2004). The form of the analysis is determined by the specific qualitative approach taken (field study, ethnography content analysis, oral history, biography, unobtrusive research) and the form of the data (field notes, documents, audiotape, videotape).

An essential component of ensuring data integrity is the accurate and appropriate analysis of research findings. Improper statistical analyses distort scientific findings, mislead casual readers (Shepard, 2002), and may negatively influence the public perception of research. Integrity issues are just as relevant to analysis of non-statistical data as well. In order to analyse data in this qualitative research, the researcher will use of thematic analysis, an analysis software package. Although some researchers suggest that disassembling, coding, and then sorting and sifting through the data, is the primary path to analysing data / data analysis. But as other rightly caution, intensive data coding, disassembly, sorting, and sifting, is neither the only way to analyse the data nor is it necessarily the most appropriate strategy.

It has been argued that they also fit the notice, collect, and think process invariably also belonging to the data analysis process.

1.8.2 ETHICAL CONSIDERATIONS

Ethics refers to the appropriateness of the researcher's behavior in relation to the rights of the participants of the study (Saunders, Lewis & Thornhill, 2009). The researcher must strike the balance between the quest for information and the rights of the participants. During information gathering, participants must be made aware of how data will be collected, analysed and reported.

The sensitivity of source code and software applications is very high and need to be protected as it an intellectual property to the organisation. The researcher was obliged to make sure that participants will not be harmed by their participation in the study. During the interviews it was necessary to hide the identity of participants of the study to preserve confidentiality.

The statement to acknowledge the consent of the participants was read at the beginning of the interview. It was also attached to the questionnaire as the cover page. In data analysis, participants were identified using the codes. The participants had the right to refuse to answer certain interview questions.

1.9 JUSTIFICATION OF THE STUDY

As team's design, develop and deploy software, it is common for multiple versions of the same software to be deployed in different sites and for the software's developers to be working simultaneously on updates. Bugs or features of the software are often only present in certain versions (because of the fixing of some problems and the introduction of others as the program develops). Therefore, for the purposes of locating and fixing bugs, it is vitally important to be able to retrieve and run different versions of the software to determine in which version(s) the problem occurs. It may also be necessary to develop two versions of the software concurrently (for instance, where one version has bugs fixed, but no new features (branch), while the other version is where new features are worked on (trunk).

Keeping track of changes has become an essential part of the practice of software development. The modern complexities of the craft are such that programmers must manage source code not only spatially, in terms of files and directory structures, but also temporally, over complex, diverging and converging timelines resulting from multiple strands of work done in parallel. Due to the many benefits such as collaboration, storing versions, restoring to previous versions, backup of using version control, there is great need to lay out a framework for the best practice for implementing **version control** in modern enterprise java web applications.

1.10 SUMMARY AND CONCLUSION

Ultimately no version control system offers an entirely satisfying answer. But each provides insight into the ins and outs of the various ways of tackling the problem of maintaining versions in software development, with a sort of concreteness and pragmatism not ordinarily seen in purely academic discussion of the subject. Moreover, along the way we get glimpses of the true depth of the problem, and why the present state of computer science and related fields is not well positioned to solving it.

Certainly, framing the challenges faced by version control systems in these terms does not tell the whole story. It might not even be the right story. But at the very least, it is an unorthodox one. Software developers rarely think about the deeper conceptual issues that underlie their tools and practices, and philosophers rarely turn their abstract ideas into concrete, practical tools. Bridging that gap is the subject of the work to follow, with the hope that it might provide insight in both directions.

CHAPTER TWO: LITERATURE REVIEW

2.1 INTRODUCTION

The previous chapter presented the introduction and the background of the study. A research problem statement was developed which led to the formulation of specific research objectives and corresponding questions. This chapter will present relevant literature to this study based on the research questions defined in the previous chapter. Okoli & Schabram (2010) on their study "identified answering of specific research question as one of the reasons for conducting literature reviews. This chapter begins by discussing the evolution of VCS, then a look at the work done by different authors as a motivation for the adoption of version control and the challenges faced, in conclusion a framework is formulated which will be used as a guide in this study.

2.2 PURPOSE OF VERSION CONTROL SYSTEMS

An important problem in program development and maintenance is version control, i.e. the task of keeping a software system consisting of many versions and configurations well organized. Below I have outlined the purpose of Version Control Systems.

- Version control enables multiple people to simultaneously work on a single project. Each person edits his or her own copy of the files and chooses when to share those changes with the rest of the team. Thus, temporary or partial edits by one person do not interfere with another person's work.
- Version control also enables one person to use multiple computers to work on a project, so it is valuable even if you are working by yourself.
- Version control integrates work done simultaneously by different team members. In most cases, edits to different files or even the same file can be combined without losing any work. In rare cases, when two people make conflicting edits to the same line of a file, then the version control system requests human assistance in deciding what to do.
- Version control gives access to historical versions of a project. This is insurance against computer crashes or data lossage. If you make a mistake, you can roll back to a previous version. You can reproduce and understand a bug report on a past version of your software. You can also undo specific edits without losing all the work that

was done in the meanwhile. For any part of a file, you can determine when, why, and by whom it was ever edited.

To gain more detailed understanding on VCS, we need to look at a brief history of the evolution of VCSs.

2.3 EVOLUTION OF VERSION CONTROL SYSTEMS

This section mainly focuses on the evolution of version control systems particularly in software development. The trajectory of version control systems can roughly be divided into three different generations, 1st generation, 2nd generation and 3rd generation. First version control systems were developed in the early 1970's when Source code control system (SCSS) was released. It was developed to help programmers control changes in the source code currently in development (Rocking, 1975, p.364).

Since then there have been a great number of version control systems and the current systems in daily use represent the 3rd generation of the software. The era of 1st generation systems spanned from early 1970's to mid-1980's. Typical for these systems were that the software stored data locally on computer and used locking method as a conflict resolution. The 2nd generation systems started to arise at the mid 1980's, replacing the old 1st generation software.

The era of the 2nd generation VCS continued till 1999. First feature that the 2nd generation systems introduced, were a centralized client -server data model which required an active network connection to function properly. The usage of the 1st generation software required that all developers of a project had to be on the same machine as the single central project repository, and the 2nd generation software changed that by allowing developers to access the single repository from another machine over the network. The second introduced feature was merge before commit as a conflict resolution. In the 1st generation software, the files were in read only format and when someone wanted to edit a file the system would make the file writable and lock it, so that no one else could edit it at the same time. In the 2nd generation software, the system noticed when a file has been changed during the time a person has been editing it and requires that the conflict is resolved before the file can be saved. The reason for this is that someone else edited the file at the same time and saved the file (Raymond, 2007; Rupelian, 2010, p. 5). The 3rd and the current generation of VCS began in 2000, when

Bitkeeper was released. Since then, a number of different version control systems have been released which all have some different functionalities and target audiences. The biggest overhaul when comparing 2nd and 3rd generation systems is the decentralized data model. In a decentralized repository model, both, the server and the client computer have a version of the repository. As the client computer has also an offline version of the repository, files and documents stored on the local repository can be edited even when the computer does not have access to the network.

The changes can be committed to the repository on a server when the computer is connected to the network. The conflict resolution method as also improved between generations. When the 1st generation systems locked the file when a user was editing it, the newer systems allow multiple users to access and edit the same document the same time. In multiple edit scenarios, when user tries to commit changes to the repository at the server, the system will inform user that the repository on the server has already been edited and the system will allow user to merge the changes he has done to the repository on the server (Sink, 2011, p. 1). A list of some of the most notable version control systems can be seen in the table below.

System name	Release year	Conflict resolution	Model
SCCS	1972	Locking	Local access
RCS	1982	Locking	Local access
CVS	1985	Merge-before-commit	Centralized client-server
Subversion	2000	Merge or locking	Distributed
Bitkeeper	2000	Commit-before-merge	Distributed
Mercurial	2005	Merge	Distributed
GIT	2005	Merge	Distributed

Table 1 2.1: Notable VCS releases (Chacon & Straub, 2014; Raymond, 2007; Tutorial Conflict, 2013)

Table 2.1 presents some of the most notable version control systems and the year were released. The table also shows the evolution of conflict Raymond resolution method, the early systems prevented conflicts by locking the edited file for one editor, current systems have built-in merge functions which can be used in a case of conflict resolution

The section above presented the brief history on VCS. Below the researcher will have a look at the various models.

2.3.1 VERSION CONTROL TYPES

Source code and other resources like icons, configuration files or documentation are the core assets of a software project (Alexander Schatten, 1996). Hence careful management of these resources is an important issue, particularly in team collaboration. The following aspects have to be taken into consideration:

- Source code should be versioned, to be able to undo changes and refer back to older version of the software
- Often older versions of a Software have to be maintained although newer version is already available (e.g. to patch security issues: Version 1, Version 2, Version 1.1, 1.2, 2.1, 2.2).
- Team-collaboration has to be considered: sharing of code between developers has to be transparent, reliable and traceable.
- Versions and other important milestones in the project should be marked in the version history to be able to go back to a specific version in the history.
- Changes in the source code should be communicated and annotated to be transparent for the whole team

Three different approaches to SCM can be distinguished: (1) local, (2) centralised and (2) distributed systems:

2.3.2 LOCAL VERSION CONTROL SYSTEMS

Depending on a software development company's need to track record of their files, the version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to. To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control. (Scott Chacon, 2009)

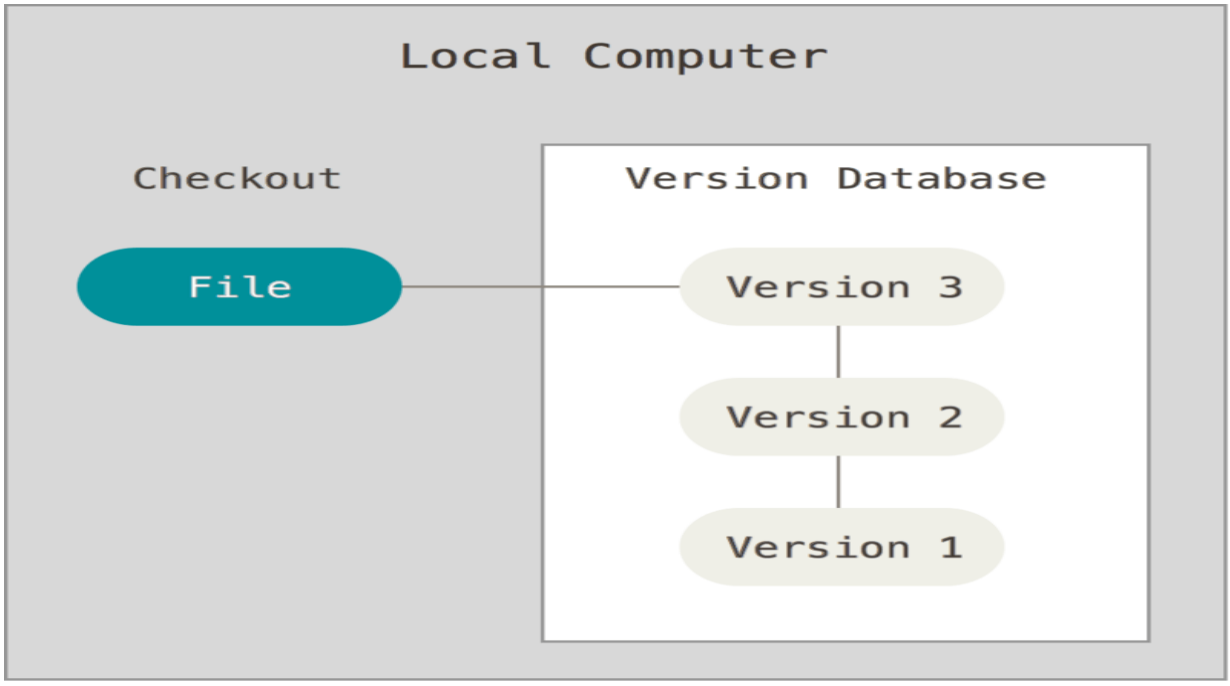

Figure 2.2: Local version control.
Source:(Ben Straub,2016)

2.3.2 CENTRALIZED VERSION CONTROL SYSTEMS

The next major issue that programs encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were developed. These systems, such as CVS, Subversion, and Perforce, have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.(Chacon & Straub, 2014)

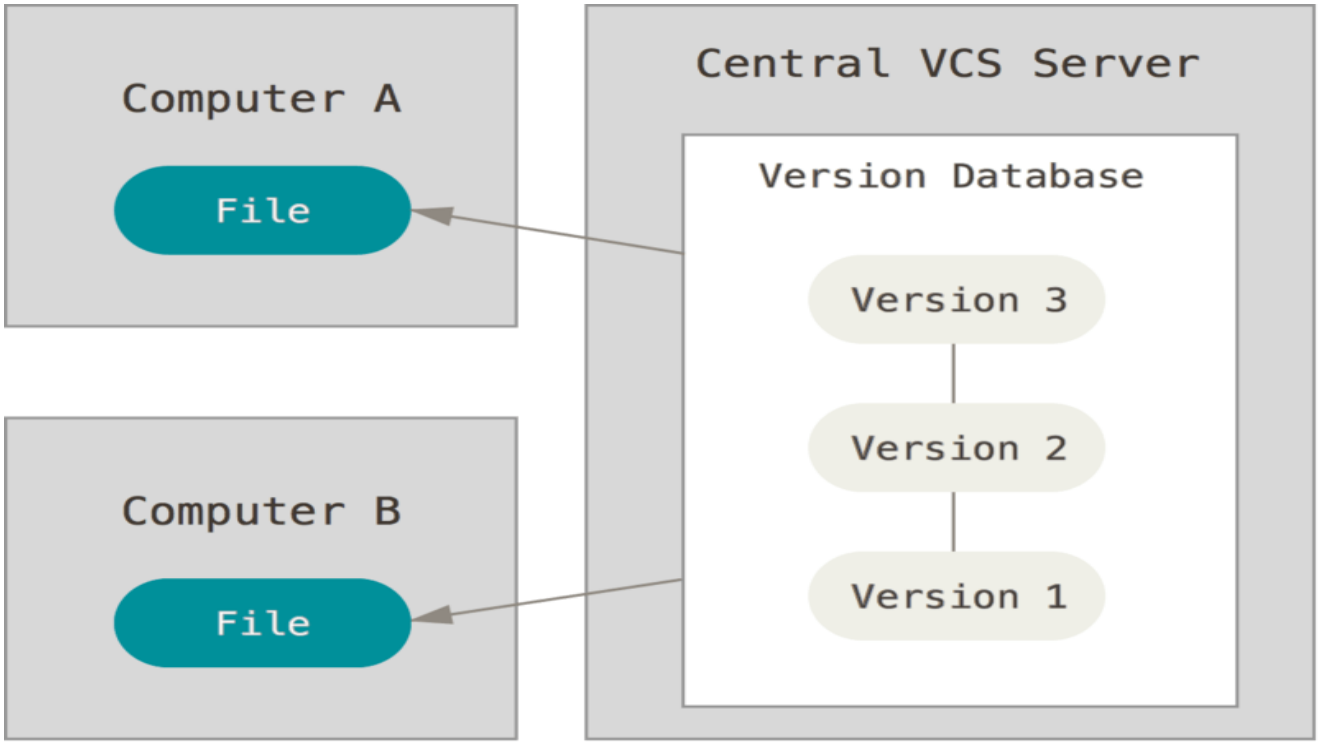

Figure 2.3 Centralized version control.
Source:(Ben Straub,2016)

This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what, and it's far easier to administer a CVCS than it is to deal with local databases on every client.

However, this setup also has some serious downsides. The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they're working on. If the hard disk the central database is on becomes corrupted, and proper backups haven't been kept, you lose absolutely everything — the entire history of the project except whatever single snapshots people happen to have on their local machines. Local VCS systems suffer from this same problem — whenever you have the entire history of the project in a single place, you risk losing everything.(Perez De Rosso & Jackson, 2013)

2.3.3 DISTRIBUTED VERSION CONTROL SYSTEMS

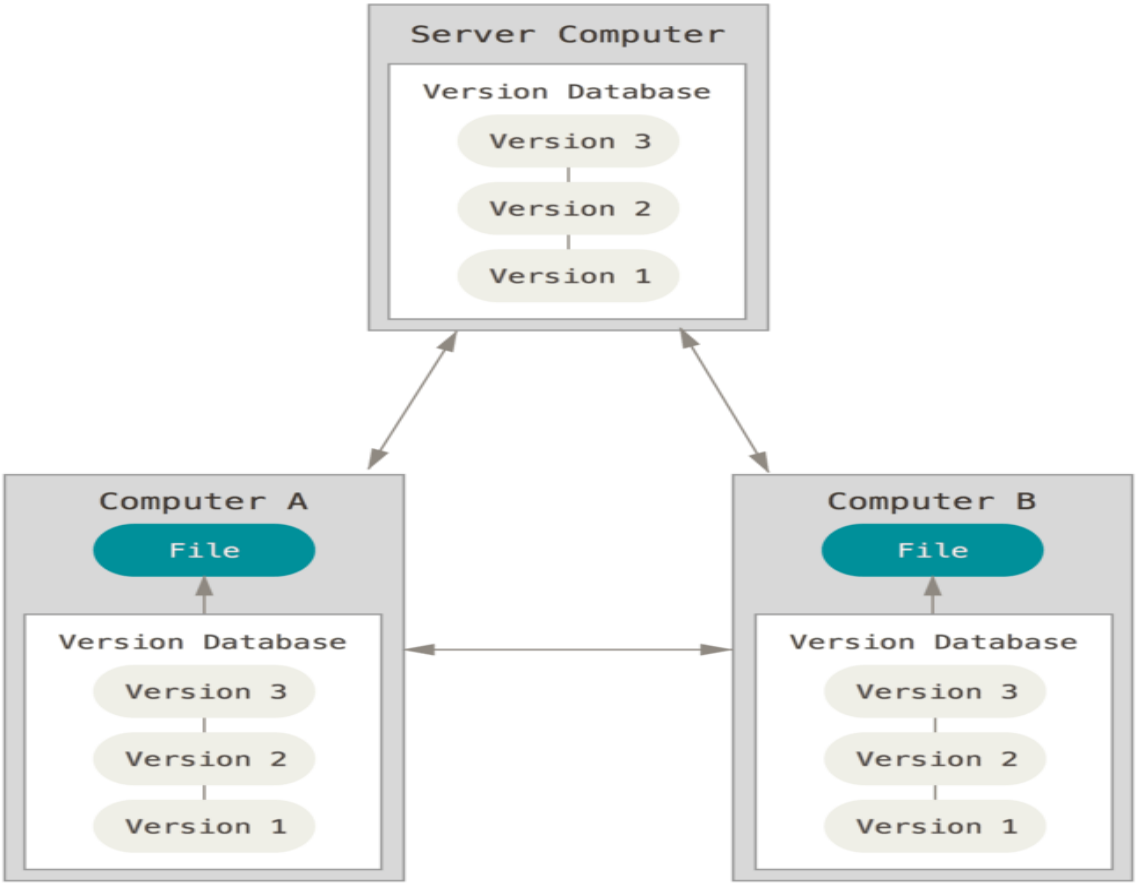

Figure 2.4: Distributed version control
Source:(Ben Straub,2016)

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don't just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history. Thus, if any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.(Chacon & Straub, 2014)

Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren't possible in centralised systems, such as hierarchical models. This research will go on to look at how VCS resolve conflicts.

2.4 WAYS TO RESOLVE CONFLICTS IN FILES

(Stefan Otte,2012) states that a version control system must have some mechanism to prevent or resolve conflicts among users who want to change the same file. There are three distinct ways to accomplish this.

2.4.1 LOCKING

The classic approach in scenarios that contain concurrency is the lock-modify-unlock mechanism. It is often called pessimistic approach. A resource gets explicitly locked by a process. Only this process can modify the resource. After modifying the resource, the lock gets removed. It guarantees, that no merging is needed, because a file can be checked out by only one user. With version-control *locking*, work files are normally read-only so that you cannot change them. You ask the VCS to make a work file writable for you by locking it; only one user can do this at any given time. When you check in your changes, that unlocks the file, making the work file read-only again. This allows other users to lock the file to make further changes. (Stefan Otte, Version Control Systems,2009)

When a locking strategy is working well, workflow looks something like this:

- Alice locks the file ErrorController.java and begins to modify it.
- Bob, attempting to modify ErrorController.java, is notified that Alice has a lock on it and he cannot check it out.
- Bob is blocked and cannot proceed. He wanders off to have a cup of coffee.
- Alice finishes her changes and commits them, unlocking ErrorController.java.
- Bob finishes his coffee, returns, and checks out ErrorController.java, locking it.

Unfortunately, workflow too often looks more like this:

- Alice locks the file ErrorController.java and begins to modify it.
- Bob, attempting to modify ErrorController.java, is notified that Alice has a lock on it and he cannot check it out.
- Alice gets a reminder that she is late for a meeting and rushes off to it, leaving ErrorController.java locked.
- Bob, attempting to modify ErrorController.java, is notified that Alice has a lock on it and he cannot check it out.

- Bob, having been thwarted twice and wasted a significant fraction of his day waiting on the lock, curses feelingly at Alice. He informs the VCS he wants to *steal the lock*.
- Alice returns from the meeting to find mail or an instant message informing her that Bob has stolen her lock.
- Changes in Alice's working copy are now in conflict with Rob's and will have to be merged later. Locking has proven useless.

That, unfortunately, is the *least* nasty failure case. If the VCS has no facility for stealing locks, change conflict is prevented but Bob may be blocked indefinitely by Alice's forgotten lock. If Alice is not reliably notified that her lock has been stolen, she may continue working on ErrorController.java only to receive a rude surprise when she attempts to commit it.

Most importantly, in these failure cases locking only defers conflicts that must be resolved by merging divergent changes to ErrorController.java after the fact — it does not prevent them. It scales poorly and tends to frustrate all parties.

Despite these problems, locking can be necessary and appropriate when conflict merging is effectively impossible. This is often the case when the work file is something non-textual, such as an image. To handle that case, some VCSes have optional locking, without a lock-stealing escape mechanism) as a fall-back, that must be explicitly requested by the user rather than being done implicitly on checkout.

Historically, locking was the first conflict-resolution method to be invented and is associated with first-generation centralized VCSes supporting local access only.

2.4.2 MERGE BEFORE COMMIT

In a *merge-before-commit* system, the VCS notices when you are attempting a commit against a file or files that have changed since you started editing and requires you to resolve the conflict before you can complete the commit.

Workflow usually looks something like this:

- Alice checks out a copy of the file ErrorController.java and begins to modify it.
- Bob checks out a copy of the file ErrorController.java and begins to modify it.
- Alice finishes her changes and commits them.

- Bob, attempting to commit the file, is informed that the repository version has changed since his checkout and he must resolve the conflict before his commit can proceed.
- Bob runs a merge command that applies Alice's changes to his working copy.
- Alice's and Bob's changes to ErrorController.java do not actually overlap. The merge command returns success, and the VCS allows Bob to commit the merged version.

Empirically, actual merge conflicts are unusual. When they do occur, a merge-before-commit VCS will typically put both spans of conflicting lines in Bob's work file with some kind of conspicuous marker around them and refuse to accept a commit until Bob has edited out the marker.

Historically, merge-before-commit was the second conflict-resolution method to be invented and is associated with centralized client-server VCSes.

2.4.3 COMMIT BEFORE MERGE

It is possible to design a VCS so it never blocks a commit. Instead, if the repository copy has changed since the files(s) were checked out, the commit can simply be shunted to a new branch. Subsequently, the branches may remain separate; or, any developer may perform a merge that brings them back together.

- Alice checks out a copy of the file ErrorController.java and begins to modify it. For illustrative purposes, let's say the revision before Alice's work file was numbered 2. The state of the project now looks like this:

Figure 2.5 commit before merge

The boxes indicate versions of ErrorController.java, the solid arrow indicates the “is a revision of” relationship in the repository, and the dotted arrow represents the checkout to Alice's working area.

- Bob checks out a copy of the file ErrorController.java and begins to modify it. The repository now looks like this:

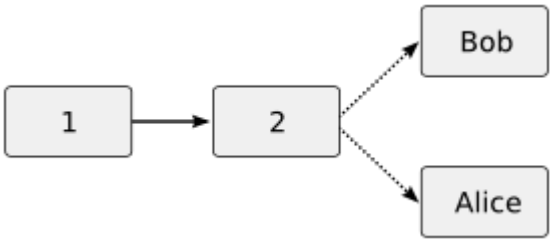

Figure 2.6 commit before merge

- Alice finishes her changes and commits them. The repository now has a version 3.

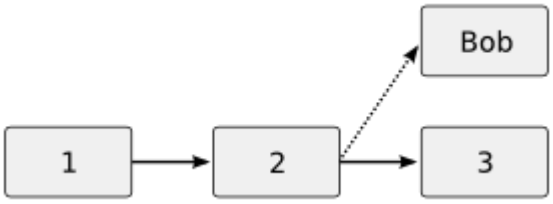

Figure 2.7 commit before merge

- Bob, attempting to commit changes to ErrorController.java, is informed that the repository version has changed since his checkout and his commit began a new branch. His revision is numbered 4. Alice and Bob now have peer branches in the repository.

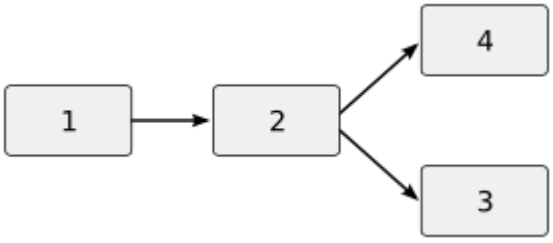

Figure 2.8 commit before merge

- Bob compares Alice's branch with his and decides the changes should be merged. He runs a merge tool to do so. When he successfully exits the merge tool, a new version of ErrorController.java is committed that is marked as a descendant of both 3 (Alice's version) and 4 (Bob's version).

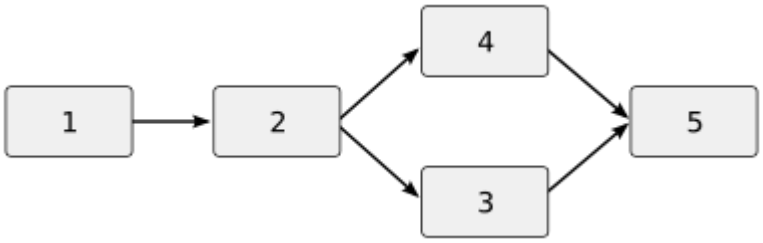

Figure 2.10: commit before merge

If a third developer, Carol, checks out ErrorController.java after Bob's merge, she will retrieve the merged version.

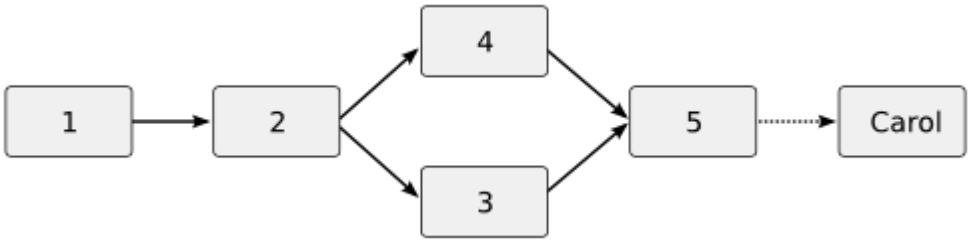

Figure 2.10: commit before merge

The commit-before-merge model leads to a very fluid development style in which developers create and converge lots of small branches. It makes experimentation easy, and records everything developers try out in a way that can be advantageous for code reviewing.

In the most general case, a repository managed by a commit-before-merge VCS can have the shape of an arbitrarily complex directed acyclic graph (DAG). Such VCSes often do *history-aware merging*, using algorithms that try to take account not just of the contents of the versions viewing merged but of the contents of their common ancestors in the repository DAG.

The model is not completely without own pitfalls, however. If another developer, David, decides to try merging revisions 3 and 4 at the same time as Bob, he might produce a different merge. Supposing David's merge is finished after Bob's, it will become version 6 and the repository will look like this:

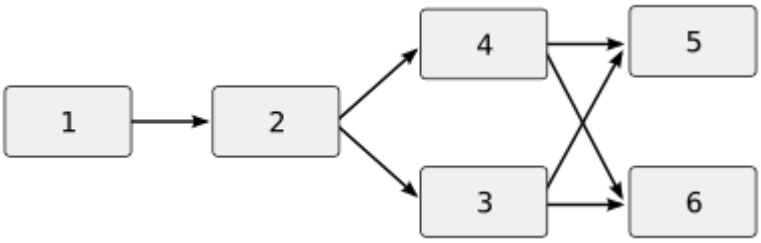

Figure 2.12: commit before merge

The variant versions 5 and 6 may now need to be reconciled. This leads to an awkward case called a *cross-merge* which tends to confuse history-aware merging tools.

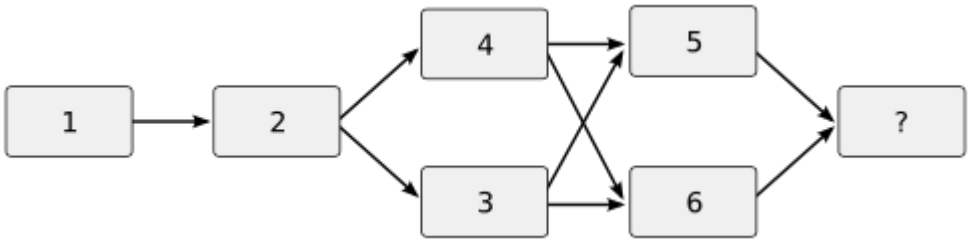

Figure 2.12: commit before merge

The commit-before-merge model is associated with third-generation decentralized VCSes and quite new, having emerged about 1998.

2.5 FILE SET OPERATIONS.

In early VCSes, some of which are still in use today, check-ins and other operations are *file-based*; each file has its own *master file* with its own comment- and revision history separate from that of all other files in the system. Later systems do *file set operations*; most importantly, a checking may include changes to several files and that change set is treated as a unit by the system. Any comment associated with the change doesn't belong to any one file but is attached to the combination of that file set and its revision number.

File sets are necessary; when a change to multiple files has to be backed out, it's important to be able to easily identify and remove all of it.

2.5.1 CONTAINER IDENTITY

(Rigbyetal.2013; Spinellis.2012) In a VCS with container identity, files and directories have a stable internal identity, initialized when the file is added to the repository. This identity

follows them through renames and moves. This means that filenames and directories are versioned, so that it is possible (for example) for the VCS to notice while doing a merge between branches that two files with different names are descendants of the same file and do a smarter merge of their contents.

Container identity can be implemented by giving each file in the repository a “true name” analogous to a Unix inode, Alternatively, it can be implemented implicitly by keeping all records of renames in history and chasing through them each time the VCS needs to check what file X was called in revision Y.

Absence of container identity has the symptom that file rename/move operations have to be modelled as a file add followed by a delete, with the deleted file's history magically copied during the add.

Usually VCSes that lack container identity also creates parent directories on the fly whenever a file is added or checked out, and you cannot actually have an empty directory in a repository.

2.5.2 SNAPSHOTS VS. CHANGESETS

(Canfora, Cerulo, & Di Penta, 2009).There are two ways to model the history of a line of development. One is as a series of snapshots of an evolving tree of files. The other is as a series of changesets transforming that tree from a root state (usually empty) to a tip state. Change comments, and metadata associated with them such as the author of the change and a timestamp, may be associated either with a snapshot or with the changeset immediately before it. From a user's point of view, the difference is indistinguishable.

CHANGESETS

Changeset-based systems have some further distinctions based on what kinds of data a changeset carries. At minimum, a changeset is a group of deltas to individual files, but there are variations in what kind of file-tree operations are represented in changesets. Changesets which include an explicit representation of file/directory moves and renames make it easy to implement container identity. (Container identity could also be implemented as a separate sequence of transaction records running parallel to a snapshot-sequence representation. As a lesser issue, a changeset-based system may either model file additions as a delta from the

empty file and file deletions as a delta to an empty file, or the changeset may explicitly record adds and deletes.

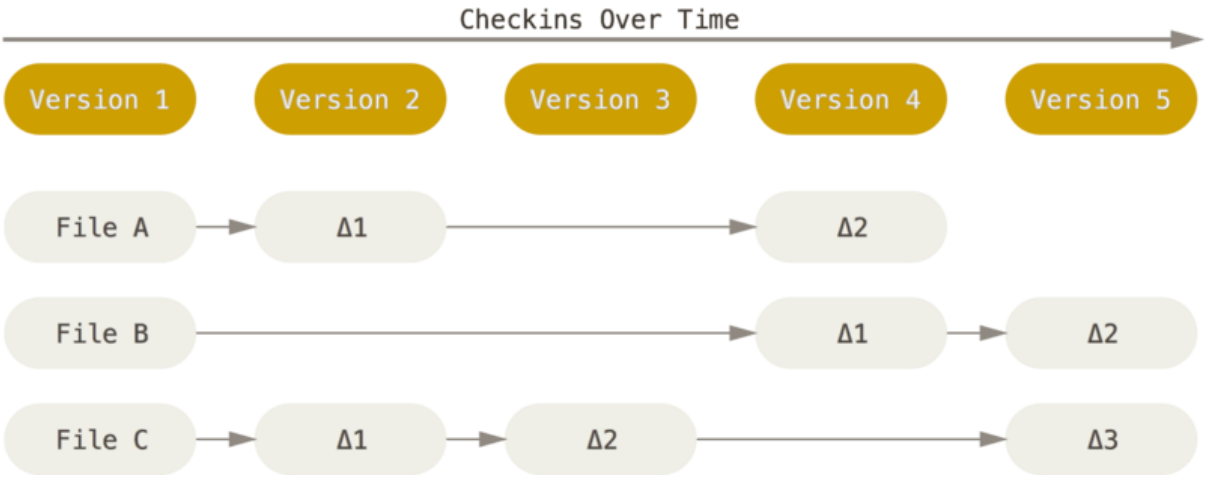

Figure 2.14: Storing data as changes to a base version of each file.

SNAPSHOTS

Snapshot and changesets are not perfectly dual representations. In a snapshot VCS it thinks of its data more like a series of snapshots of a miniature filesystem. With snapshot VCS, every time you commit, or save the state of your project, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, snapshot VCS doesn't store the file again, just a link to the previous identical file it has already stored. it thinks about its data more like a **stream of snapshots**. This is an important distinction between Git and nearly all other VCSs. It makes Git reconsider almost every aspect of version control that most other systems copied from the previous generation. This makes Git more like a mini filesystem with some incredibly powerful tools built on top of it, rather than simply a VCS.

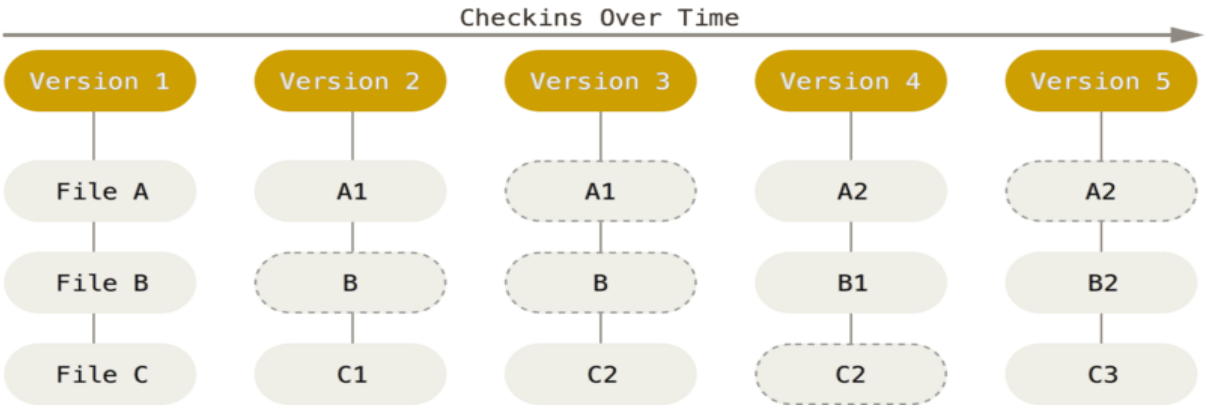

Figure 2.5. Storing data as snapshots of the project over time.

2.6 BASICS CONCEPTS OF VCS FRAMEWORKS.

In order to come up with a framework, there is need to understand the functionalities of at least 3 version control systems in order to gain an understanding of their operations. I will look at the common features of Versioning Systems. The following paragraphs, outline the various components of version control systems. This section is a concise and informative compendium that serves as both an introduction and a manual for practical usage of Mercurial, Git, Veracity version control system. On all the names VCS, they have these common features:

2.6.1.2 CREATING A NEW, EMPTY REPOSITORY

(Sink,(2011), A repository is the official place where you store all your work. It keeps track of your tree, by which I mean all your files, as well as the layout of the directories in which they are stored. A repository contains history. A file system is two-dimensional: Its space is defined by directories and files. In contrast, a repository is three-dimensional: It exists in a continuum defined by directories, files, and time. A version control repository contains every version of your source code that has ever existed.

A consequence of this idea is that nothing is ever really destroyed. Every time you make some kind of change to your repository, even if that change is to delete something, the repository gets larger because the history is longer. Each change adds to the history of the repository. We never subtract anything from that history.

The create operation is used to create a new repository. This is one of the first operations you will use, and after that, it gets used a lot less often. When you create a new repository, your VCS will expect you to say something to identify it, such as where you want it to be created, or what its name should be.

2.6.1.3 CHECKOUT

The checkout operation is used when you need to make a new working copy for a repository that already exists. A working copy is a snapshot of the repository used by a developer as a place to make changes. The repository is shared by the whole team, but people do not modify it directly. Rather, each individual developer works by using a working copy. The working copy provides her with a private workspace where she can do her work isolated from the rest of the team. Let’s imagine for a moment what life would be like without this distinction between working copy and repo-sitory. In a single-person team, the situation could be described as tolerable. However, for any number of developers greater than one, things can

get very messy. I've seen people try it. They store their code on a file server. Everyone uses network file sharing and edits the source files in place. When somebody wants to edit main.cpp, they shout across the hall and ask if anybody else is using that file. (Sink, 2011) Their Ethernet is saturated most of the time because the developers are actually compiling on their network drives.

With a version control tool, working on a multi-person team is much simpler. Each developer has a working copy to use as a private workspace. He can make changes to his own working copy without adversely affecting the rest of the team. The working copy is actually more than just a snapshot of the contents of the repository. It also contains some metadata so that it can keep careful track of the state of things.

For housekeeping purposes, the version control tool usually keeps a bit of extra information with the working copy. When a file is retrieved, the VCS stores its contents in the corresponding working copy of that file, but it also records certain information. For example:

2.6.1.4 COMMIT

Apply the modifications in the working copy to the repository as a new changeset. This is the operation that actually modifies the repository. Several others modify the working copy and add an operation to a list we call the pending changeset, a place where changes wait to be committed. The commit operation takes the pending changeset and uses it to create a new version of the tree in the repository.

All modern version control tools perform this operation atomically. In other words, no matter how many individual modifications are in your pending changeset, the repository will either end up with all of them (if the operation is successful), or none of them (if the operation fails). It is impossible for the repository to end up in a state with only half of the operations done. The integrity of the repository is assured.

(Sink, 2011)It is typical to provide a log message (or comment) when you commit, explaining the changes you have made. This log message becomes part of the history of the repository.

2.6.1.5 UPDATE THE WORKING COPY

Update brings your working copy up-to-date by applying changes from the repository, merging them with any changes you have made to your working copy if necessary. When the working copy was first created, its contents exactly reflected a specific revision of the repository. The VCS remembers that revision so that it can keep careful track of where you

started making your changes. This revision is often referred to as the parent of the working copy, because if you commit changes from the working copy, that revision will be the parent of the new changeset **Update** is sort of like the mirror image of **commit**. Both operations move changes between the working copy and the repository. **Commit** goes from the working copy to the (Sink, 2011)repository. **Update** goes in the other direction.

2.6.1.6 ADD A FILE OR DIRECTORY.

Use the add operation when you have a file or directory in your working copy that is not yet under version control and you want to add it to the repository. The item is not actually added immediately. Rather, the item becomes part of the pending changeset, and is added to the repository when you commit.

2.6.1.7 MODIFYING A FILE

This is the most common operation when using a version control system. When you checkout, your working copy contains a bunch of files from the repository. You modify those files, expecting to make your changes a part of the repository. With most version control tools, the edit operation doesn't actually involve the VCS directly. You simply edit the file using your favourite text editor or development environment and the VCS will notice the change and make the modified file part of the pending changeset. On the other hand, some version control tools want you to be more explicit. Such tools usually set the filesystem read-only bit on all files in the working copy. Later, when you notify the VCS that you want to modify a file, it will make the working copy of that file writable.

2.6.1.8 DELETE A FILE OR DIRECTORY.

Use the delete operation when you want to remove a file or directory from the repository. If you try to delete a file which has been modified in your working copy, your VCS might complain. Typically, the delete operation will immediately delete the working copy of the file, but the actual deletion of the file in the repository is simply added to the pending changeset. Recall that in the repository the file is not really deleted. When you commit a changeset containing a delete, you are simply creating a new version of the tree which does not contain the deleted file. The previous version of the tree is still in the repository, and that version still contains the file.

2.6.1.9 RENAME A FILE OR DIRECTORY

Use the rename operation when you want to change the name of a file or directory. The operation is added to the pending changeset, but the item in the working copy typically gets renamed immediately. There is lot of variety in how version control tools support renames. Some of the earlier tools had no support for rename at all. Some tools (including Bazaar and Veracity) implement rename formally, requiring that they be notified explicitly when something is to be renamed. Such tools treat the name of a file or directory as simply one of its attributes, subject to change over time. Still other tools (including Git) implement rename informally, detecting renames by observing changes rather than by keeping track of the identity of a file. Rename detection usually works well in practice, but if a file has been both renamed and modified, there is a chance the VCS will do the wrong thing.

2.6.1.10 MOVE A FILE OR DIRECTORY.

Use the move operation when you want to move a file or directory from one place in the tree to another. The operation is added to the pending changeset, but the item in the working copy typically gets moved immediately. Some tools treat renames and move as the same operation (in the Unix tradition of treating the file's entire path as its name), while others keep them separate (by thinking of the file's name and its containing directory as separate attributes).

2.6.1.11 STATUS

As you make changes in your working copy, each change is added to the pending changeset. The status operation is used to see the pending change-set. Or to put it another way, status shows you what changes would be applied to the repository if you were to commit.

2.6.1.12 DIFF

Status provides a list of changes but no details about them. To see exactly what changes have been made to the files, you need to use the diff operation. Your VCS may implement diff in a number of different ways. For a command-line application, it may simply print out a diff to the console. Or your VCS might launch a visual diff application. (Sink, 2011).

2.6.1.13 REVERT

Sometimes I make changes to my working copy that I simply don't intend to keep. Perhaps I tried to fix a bug and discovered that my fix introduced five new bugs which are worse than the one I started with. Or perhaps I just changed my mind. In any case, a very nice feature of

a working copy is the ability to revert the changes I have made. A complete revert of the working copy will throw away all your pending changes and return the working copy to the way it was just after you did the checkout.

2.6.1.14. LOG

Your repository keeps track of every version that has ever existed. The log operation is the way to see those records. It displays each changeset along with additional data such as

- Who made the change?
- When was the change made?
- What was the log message?

Most version control tools present ways of slicing and dicing this information. For example, you can ask log to list all the changesets made by the user named Leonardo, or all the changesets made during April 2010.

2.6.1.15 TAG

Version control tools provide a way to mark a specific instant in the history of the repository with a meaningful name. This is not altogether different from the descriptive and memorable names we use for variables and constants in our code. Which of the following two lines of code is easier to understand?

2.6.1.16 BRANCH -CREATE

The branch operation is what you use when you want your development process to fork off into two different directions. For example, when you release version 3.0, you might want to create a branch so that development of 4.0 features can be kept separate from 3.0.x bug-fixes.

2.6.1.17 MERGE

Typically, when you have used branch to enable your development to diverge, you later want it to converge again, at least partially. For example, if you created a branch for 3.0.x bug-fixes, you probably want those bug-fixes to happen in the main line of development as well. Without the merge operation, you could still achieve this by manually doing the bug-fixes in both branches. Merge makes this operation simpler by automating things as much as possible.

2.6.1.18. RESOLVE

In some cases, the merge operation requires human intervention. Merge automatically deals with everything that can be done safely. Everything else is considered a conflict. For example, what if the file foo.js was modified in one branch and deleted in the other? This

kind of situation requires a person to make the decisions. The resolve operation is used to help the user figure things out and to inform the VCS how the conflict should be handled.

2.6.1.19 LOCK

The lock operation is used to get exclusive rights to modify a file. Not all version control tools include this feature. In some cases, it is provided but is intended to be rarely used. For any files that are in a format based on plain text (source code, XML, etc.), it is usually best to just let the VCS handle the concurrency issues. But for binary files which cannot be automatically merged, it can be handy to grab a lock on a file.(Chacon & Straub, 2014)

2.7 CHALLENGES IN VERSION CONTROL ADOPTION

The decision to adopt enterprise Git as a corporate source code management (SCM) standard is not merely a technical concern. It also has a major impact on the business performance of your organization. To effectively address the challenges of enterprise Git adoption, organizations should set the bar high by handling Git as a strategic investment.

In order to be successful with implementing Git in the enterprise, be sure to select an Enterprise Version Control (EVC) platform for centralized, distributed and hybrid development environments. This platform must also integrate with existing development tools enabling Continuous Integration (CI) and Continuous Delivery (CD). (Dept. of Inf. Process. Sci., Oulu Univ., Finland,2016)

Here are the challenges of Enterprise Git adoption:

- Insufficient native access and audit controls in Git

Git was designed for the needs of open source projects, specifically Linux. It can reliably track who authored a change and who can add changes to a repository. While sufficient for most open source projects, this design consideration left out many of the inherent security controls of centralized version control tools, such as Subversion (SVN), which is currently used by the enterprise. This realization often comes as a shock to security and compliance officers in large companies once they discover the risks associated with migration from centralized legacy SCM to Git.

- Heterogeneity of modern enterprise development infrastructure

As long as hybrid SCM is still the norm, there is a need to manage Git and SVN simultaneously—not only within the enterprise but also at the individual project level. However, most Git solutions in the market today ignore this need completely and do not provide any management or governance for anything except Git itself. As a result, organizations often experience difficulties articulating their SVN/Git infrastructure coexistence strategy or face an “all or nothing” migration dilemma.

- Avoiding disruption of existing business processes

Large companies are very sensitive to “deregulation” permitted by Git’s powerful branching capabilities as it potentially may cause a lot of changes in the company’s existing software delivery lifecycle management processes, causing disruption to automated integration, build,

and delivery infrastructure investments. These concerns are legitimate, and stakeholders across the organizations need to have a means to preserve existing investments and replicating the company's current business processes.

- Need to maintain highly distributed, federated repositories

While the use of a central master repository (also referred to as a "golden," "canonical," or "blessed" repository) is easier to manage, Git does not mandate this approach. For the enterprise, however, federated deployments pose a serious risk of intellectual property or data loss. With dozens or hundreds of servers, it becomes challenging to locate code or enforce strategies for backup, disaster recovery, or failover.

2.8 CONCLUSION

A literature review was conducted based on the main research questions of the study. The common characteristics between DVCS and CVS were noted, and with these common features there comes a gap on how to best implement the best practise of VCS in their different architectures. Most research centred on presenting usage of different Version Control Systems but little to none has been written on the best practice on implementing version control but rather presenters mainly concentrated on the pros and cons of the VCSs. This current study will make effort to fill in the gap by providing a concrete framework on the best practise of implementing VCS, giving the developers flexibility in choosing any VCS of their choice but being guided by this framework on the best practises. The following chapter will look at Research Methodology.

CHAPTER THREE: RESEARCH METHODOLOGY

3.1 INTRODUCTION

A research methodology is presented in this chapter. The way in which research is conducted may be conceived of in terms of the research philosophy subscribed to, the research strategy employed and so the research instruments utilised (and perhaps developed) in the pursuit of a goal – the research objective(s) - and the quest for the solution of a problem - the research question. To further elaborate on a research methodology, Creswell (2014) defined it as a combination of elements that affect the study in a logical way to effectively address the research question. Thus, it is a well laid out plan for collecting and analysing data during a study.

This chapter presents the basic building blocks in social science research as well as a detailed argumentation for **critical realism** as the philosophical approach underpinning this study. The positivist, interpretivist and critical realist paradigms are presented and aligned with their ontological, epistemological and methodological concerns. This chapter also presents the research design including description of theoretical analyses and justification.

The research will make use of (Saunders et al,2009) research onion. The diagram shown below.

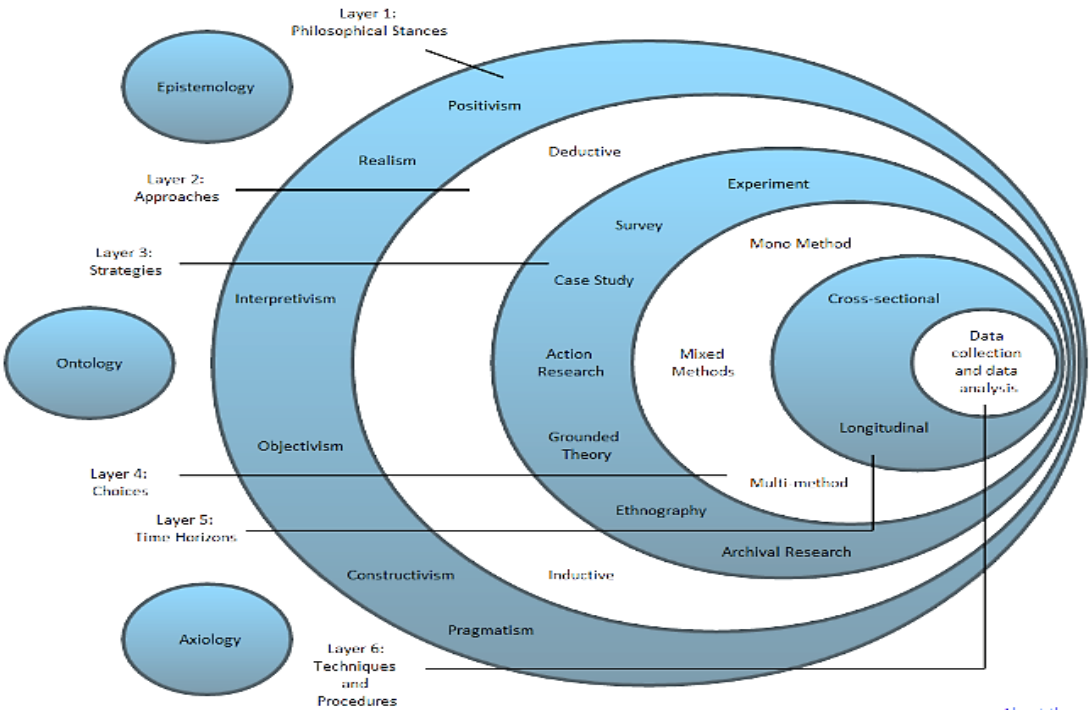

Figure 3.1: Research onion (Source: Saunders et al',2009)

3.2 RESEARCH PHILOSOPHY

A research philosophy is a belief about the way in which data about a phenomenon should be gathered, analysed and used. The term epistemology (what is known to be true) as opposed to doxology (what is believed to be true) encompasses the various philosophies of research approach. The purpose of science, then, is the process of transforming things believed into things known: doxa to episteme. From Saunders' Research onion a number of philosophies have been outlined namely Positivism, Realism, Interpretivism, Objectivism, Subjectivism, Pragmatism, Functionalism, Interpretive, radical humanist, radical structuralist Three major research philosophies have been identified in the Western tradition of science, namely positivist (sometimes called scientific), Critical Realism and interpretivist (also known as ant positivist) (Galliers, 1991).

The confidence provided by understanding different philosophical positions provides the researcher and the practitioner with the power to argue for different research approaches and allows one confidently to choose one's own sphere of activity", (Dobson, 2002).

3.2.1 POSITIVISM

Positivists believe that reality is stable and can be observed and described from an objective viewpoint (Levin, 1988), i.e. without interfering with the phenomena being studied. They contend that phenomena should be isolated and that observations should be repeatable. This often involves manipulation of reality with variations in only a single independent variable so as to identify regularities in, and to form relationships between, some of the constituent elements of the social world.

Predictions can be made on the basis of the previously observed and explained realities and their inter-relationships. "Positivism has a long and rich historical tradition. It is so embedded in our society that knowledge claims not grounded in positivist thought are simply dismissed as a scientific and therefore invalid" (Hirschheim, 1985, p.33). This view is indirectly supported by Alavi and Carlson (1992) who, in a review of 902 IS research articles, found that all the empirical studies were positivist in approach. Positivism has also had a particularly successful association with the physical and natural sciences. There has, however, been much debate on the issue of whether or not this positivist paradigm is entirely suitable for the social sciences (Hirschheim, 1985), many authors calling for a more pluralistic attitude towards IS research methodologies (e.g. Kuhn, 1970; Bjorn-Andersen, 1985; Remenyi and Williams, 1996). While I shall not elaborate on this debate further, it is

germane to our study since it is also the case that Information Systems, dealing as it does with the interaction of people and technology, is considered to be of the social sciences rather than the physical sciences (Hirschheim, 1985). Indeed, some of the difficulties experienced in IS research, such as the apparent inconsistency of results, may be attributed to the inappropriateness of the positivist paradigm for the domain. Likewise, some variables or constituent parts of reality might have been previously thought unmeasurable under the positivist paradigm - and hence went unresearched (Galliers, 1991).

3.2.2 INTERPRETIVISM

Interpretivists contend that only through the subjective interpretation of and intervention in reality can that reality be fully understood. The study of phenomena in their natural environment is key to the interpretivist philosophy, together with the acknowledgement that scientists cannot avoid affecting those phenomena they study. They admit that there may be many interpretations of reality but maintain that these interpretations are in themselves a part of the scientific knowledge they are pursuing. Interpretivism has a tradition that is no less glorious than that of positivism, nor is it shorter.

3.2.3 CRITICAL REALISM

Several surveys of information systems literature (Mingers, 2004; Walsham, 1995, Olikowski and Baroudi, 1991) demonstrate the dominance of positivist and social constructivist philosophies for many years. However, with the emergence of contemporary trans-paradigmatic in information systems research, critical realism is gaining space as a fundamental research philosophy. While other philosophical assumptions in IS research are not preferred in this research, it is however deemed necessary to give their overview leading to the argumentation which positions this research within the critical realist paradigm.

Critical realism has emerged as a philosophical stance in research which often combines a realist ontological perspective (theory of being) with a relativist epistemology (theory of knowledge). Since the 18th century, positivist paradigm has provided the philosophical underpinnings of mainstream scientific research. However, the shortcomings of positivism are well recognised in much of the contemporary literature, on one hand. The application of positivist precepts to research on social phenomena is problematic (Evered and Louis, 1981; Galliers and Land, 1987; Guba and Lincoln, 1985; Morgan, 1980; Morgan and Smircich, 1980; Weick, 1984) (as cited by Orlikowski and Baroudi (1991)). On the other hand, the

primary limitation to interpretive research is that it abandons the scientific procedures of verification and therefore results cannot be generalized to other situations.

Further, the ontological assumption of interpretivism is subjective rather than objective but critical realism posits that reality exists that is not contingent on human perception. The inherent gap created by positivism and interpretivism is covered by the critical realist paradigm. According to Mingers (2004), the major advantage of a critical realist approach is that it maintains reality whilst still recognizing the inherent meaningfulness of social interaction.

CRITICAL REALIST ONTOLOGY

According to Bhaskar (1975), critical realism is a realist theory of science. Bhaskar asserts that critical realism is:

(a) kind of ontology in which the world was seen as structured, differentiated and changing. And science was seen as a process in motion attempting to capture ever deeper and more basic strata of a reality at any moment of time unknown to us and perhaps not even empirically manifest. Structures are changing, differentiated" (Baskhar,2017).

Bhaskar (1978) asserts that ontology in critical realism takes priority over epistemology, suggesting that it is the nature of the scientific object that should determine its proper epistemology. The identification of "epistemic fallacy" is a critical weapon against previously dominant traditions. Epistemic fallacy is elaborated by Mingers in the following statements:

"… reducing the ontological domain of existence to the epistemological domain of knowledge—statements about being are translated into ones about our (human) knowledge or experience of being. For the empiricist, that which cannot be experienced cannot be. For the conventionalist, limitations of our knowledge of being are taken to be limitations on being itself. In contrast, the realist asserts the primacy of ontology—the world would exist whether or not humans did" (Mingers, 2004).

Arguing against epistemic fallacy, Bhaskar (1978:39: asserts that:

"knowledge follows existence, in logic and in time; and any philosophical position which explicitly or implicitly denies this has got things upside down"

Critical realism is predicated on the manifesto to recognize the reality of the natural order and the events and discourses of the social world (Bhaskar, 1989 as cited by Carlsson, 2000). Bhaskar identifies three domains (Figure 3.2): the real, the actual, and the empirical. The events and discourses which Bhasker claims will help us understand the world are generated by the structures and mechanisms. The observable experiences (the empirical) are contained in the actual events which have been generated by mechanisms while the mechanisms that have generated the actual events (the Real) embody both the actual and the empirical.

CRITICAL REALIST EPISTEMOLOGY

For knowledge to be gained, every theory of knowledge (epistemology) must presuppose the existence of a theory of what the world looks like (ontology) (Patomaki and Wight, 2000). The epistemological position of critical realism is therefore that science is composed of two dimensions: the intransitive and the transitive. The intransitive is the object of scientific inquiry and the transitive is our conceptions of that object. The relevance of knowledge is dependent on the nature, power and mechanism of the objective reality (Alvesson et al, 2000). While the intrusive dimension is relatively enduring, the transitive changes under different contexts. This notion is elaborated by Danermark et al (2002:26):

"While it is evident that reality exists and is what it is, independently of our knowledge of it, it is also evident that the kind of knowledge that is produced depends on what problems we have and what questions we ask in relation to the world around us".

The fundamental epistemological tenet of critical realism is therefore that knowledge is always historically and socially located, without losing the ontological dimension. Within the social science research context, the transitive objects of science are the theories which researchers use to understand the world. The theories actually point to something external to it, a reality which is independent of the researcher. This reality which is sought to be understood and explained by the theories is the intransitive object of science.

CRITICAL REALIST METHODOLOGY

Critical realists employ a wide range of methods and their range of methodological assumptions is wider than one might expect because of their efforts to work with social theory (Olsen, 2009). The central tenet of critical realist methodology is that science is concerned with explanation, understanding and interpretation rather than discovering universal laws, predictive ability or the simple description of meanings and beliefs (Mingers,

2006). This central tenet brings methodological contribution of realism closer to interpretivism, where there is a possibility of multiple interpretations of one reality, than to empiricism.

A major methodological starting-point which is common among realists is retroduction. Retroduction is one form of transcendental realism in which researchers ask 'why things appear as they do' or 'why things are being observed as they seem to be' (Olsen, 2009). Transcendental Realism (empiricism) posits that entities and mechanisms discovered by science exist as they are regardless of human access to them and the methodological contribution to science is through empirical experimentation**.**

IMPORTANCE OF CRITICAL REALISM IN THIS RESEARCH

While it has often been observed that no single research methodology is intrinsically better than any other (Benbasat et al., 1987), and many researchers calling for a combination of research methods (Kaplan and Duchon, 1988), critical realist methodology can best be adopted for the problem under investigation in this study. This study investigates a social phenomenon which leads "to the production of knowledge that can result in emancipatory change" (Ochara, 2009). As stated earlier, the central tenet of critical realist methodology is that science is concerned with explanation, understanding and interpretation, the aim of this research is to provide a framework for the best practice in Source Code Version Control. According to the realist philosophy, the question which a researcher seeks to answer in the world of social science, of which information systems partly lies is:

"What properties do societies and people possess that might make them possible objects of knowledge for us?" (Bhaskar, 1979:17).

Many scholars assert that "information systems are fundamentally social rather than technical" (Hirschheim, 1985:1335). Morton (2006) concurs with Hirschheim and further argues that if it is accepted among systems developers that information systems are social systems, then IS can be directly situated within social sciences and critical realism provides philosophical underpinnings for such research.

A characteristic of a critical realist's view of knowledge is that it should lead to emancipatory change (Ochara, 2009).

3.3 RESEARCH APPROACH

Two types of approaches are outlined here: the deductive and the inductive approach.

3.3.1 DEDUCTIVE APPROACH

It is also called the testing theory. The deductive approach develops the hypothesis or hypotheses upon a pre-existing theory and then formulates the research approach to test it (Silverman, 2013). This approach is best suited to contexts where the research project is concerned with examining whether the observed phenomena fit with expectation based upon previous research (Wiles et al., 2011). The deductive approach thus might be considered particularly suited to the positivist approach, which permits the formulation of hypotheses and the statistical testing of expected results to an accepted level of probability (Snieder & Larner, 2009). However, a deductive approach may also be used with qualitative research techniques, though in such cases the expectations formed by pre-existing research would be formulated differently than through hypothesis testing (Saunders et al., 2007). The deductive approach is characterised as the development from general to particular: the general theory and knowledge base is first established and the specific knowledge gained from the research process is then tested against it (Kothari, 2004).

3.3.2 INDUCTIVE APPROACH

The inductive approach is characterised as a move from the specific to the general (Bryman & Bell, 2011). In this approach, the observations are the starting point for the researcher, and patterns are looked for in the data (Beiske, 2007). In this approach, there is no framework that initially informs the data collection and the research focus can thus be formed after the data has been collected (Flick, 2011). Although this may be seen as the point at which new theories are generated, it is also true that as the data is analysed that it may be found to fit into an existing theory (Bryman & Bell, 2011).

This method is more commonly used in qualitative research, where the absence of a theory informing the research process may be of benefit by reducing the potential for researcher bias in the data collection stage (Bryman & Bell, 2011). Interviews are carried out concerning specific phenomena and then the data may be examined for patterns between respondents (Flick, 2011). However, this approach may also be used effectively within positivist methodologies, where the data is analysed first and significant patterns are used to inform the generation of results. The researcher will make use of **the inductive/building** theory approach as it is associated with qualitative research design and has the following emphasis:

- gaining an understanding of the meanings humans attach to events
- a close understanding of the research context
- the collection of qualitative data
- a more flexible structure to permit changes of research emphasis as the research progresses
- a realisation that the researcher is part of the research process
- less concern with the need to generalise

3.4 RESEARCH STRATEGIES

The research strategy is how the researcher intends to carry out the work (Saunders et al., 2007). The strategy can include a number of different approaches, such as experimental research, action research, case study research, interviews, surveys, or a systematic literature review.

EXPERIMENTAL RESEARCH

Experimental research refers to the strategy of creating a research process that examines the results of an experiment against the expected results (Saunders et al., 2007). It can be used in all areas of research, and usually involves the consideration of a relatively limited number of factors (Saunders et al., 2007). The relationship between the factors are examined and judged against the expectation of the research outcomes.

CASE STUDY

Case study research is the assessment of a single unit in order to establish its key features and draw generalisations (Bryman, 2012). It can offer an insight into the specific nature of any example and can establish the importance of culture and context in differences between cases (Silverman, 2013). This form of research is effective in financial research, such as comparing

the experiences of two companies, or comparing the effect of investment in difference contexts.

GROUNDED THEORY

Grounded theory is a qualitative methodology that draws on an inductive approach whereby patterns are derived from the data as a precondition for the study (May, 2011). For example, interview data may be transcribed, coded and then grouped accordingly to the common factors exhibited between respondents. This means that the results of the research are derived fundamentally from the research that has been completed, rather than where the data is examined to establish whether it fits with pre-existing frameworks (Flick, 2011). Its use is common in the social sciences (Bryman, 2012).

SURVEYS

Surveys tend to be used in quantitative research projects and involve sampling a representative proportion of the population (Bryman & Bell, 2011). The surveys produce quantitative data that can be analysed empirically. Surveys are most commonly used to examine causative variables between different types of data.

ETHNOGRAPHY

Ethnography involves the close observation of people, examining their cultural interaction and their meaning (Bryman, 2012). In this research process, the observer conducts the research from the perspective of the people being observed and aims to understand the differences of meaning and importance or behaviours from their perspective.

AN ARCHIVAL RESEARCH STRATEGY

An archival research strategy is one where the research is conducted from existing materials (Flick, 2011). The form of research may involve a systematic literature review, where patterns of existing research are examined and summed up in order to establish the sum of knowledge on a particular study, or to examine the application of existing research to specific problems. Archival research may also refer to historical research, where a body of source material is mined in order to establish results.

ACTION RESEARCH

The researcher has chosen this research strategy as he will be actively involved in the processes. Action research is a form of applied research where the researcher attempts to

develop results or a solution that is of practical value to the people with whom the research is working, and at the same time developing theoretical knowledge. Through direct intervention in problems, the researcher aims to create practical, often emancipatory, outcomes while also aiming to rein form existing theory in the domain studied. As with case studies, action research is usually restricted to a single organisation making it difficult to generalise findings, while different researchers may interpret events differently. The personal ethics of the researcher are critical, since the opportunity for direct researcher intervention is always present.

3.5 RESEARCH DESIGN

The researcher will make use of qualitative research design. Qualitative research design according to Harrell & Bradley (2009), is an approach aimed at understanding the experiences of human beings by applying specific research methods such as interviews, observations, qualitative questionnaires and document review. Qualitative research put emphasis on words in the collection and analysis of data in order to facilitate the generation of themes and patterns emerging from a study (Williams, 2007; Owen, 2014).

The qualitative research approach was chosen to obtain views, opinions and experiences the current processes in version control systems. It also sought to understand the efforts made by the development team to implement version control at different stages of software development... The information was critical in assisting the researcher to build or develop the proposed framework.

3.6 TIME HORIZON

LONGITUDINAL OR CROSS SECTIONAL

Two types of time horizons are specified within the research onion: the cross sectional and the longitudinal (Bryman, 2012). The Time Horizon is the time framework within which the project is intended for completion (Saunders et al., 2007).

A longitudinal time horizon for data collection refers to the collection of data repeatedly over an extended period and is used where an important factor for the research is examining change over time (Goddard & Melville, 2004). This has the benefit of being used to study change and development. Furthermore, it allows the establishment of some control over the

variables being studied. The time horizon selected is not dependent on a specific research approach or methodology (Saunders et al., 2007).

The cross-sectional time horizon is one already established, whereby the data must be collected. This is dubbed the snapshot time collection, where the data is collected at a certain point (Flick, 2011). This is used when the investigation is concerned with the study of a particular phenomenon at a specific time. The researcher will make use of a cross sectional time horizon given the time frame the research is supposed to be completed.

3.7 DATA COLLECTION METHODS

Data collection stresses the development of logical research evidence that facilitate information gathering in a specific research (Ololube, Kpolovie & Harcourt, 2012). Data can be collected from both primary and secondary sources in order to provide evidence of a researcher. Yin (2014) stated that there are six (6) different sources for evidences which researchers could use during data collection in a case study research. These include documentation, archival records, interviews, direct observations, participant observation, and physical artefacts. The following data collection methods interviews, questionnaires and observation were used during the study.

"A research method is a technique for (or a way of proceeding in) gathering evidence. One could reasonable argue that all-evidence gathering techniques fall into one of the following three categories: listening to (or interrogating) informants, observing behaviour, or examining historical traces and records. In this sense, there are only three methods of social inquiry".

LITERATURE REVIEW

Literature review is the study and acknowledgement of other scholars' researches that are in line with the current study. This research instrument is commonly used in collecting secondary data. The secondary data that were included in the study were obtained from journal articles and books. A literature review was carried out to enable the researcher to establish the route of the study. Models, frameworks and methodologies that were found useful were used in this study to understand the concepts and theories required in version control systems.

INTERVIEWS

The purpose of an interview as a research instrument is to dig into the people's views and experiences about certain issues to be studied (Gill et al., 2008). They are basically grouped into three basic categories: structured, semi-structured and unstructured. Semi-structured interviews which involve a series of open-ended questions based on the topic areas the researcher wants to cover (Harrell & Bradley 2009), were used in this study. This type of the interview enabled the researcher to encourage the participants to give more details on previous responses that could have been partially answered.

In designing the interview questions, the researcher followed a general interview guide approach protocol (Bricki & Green, 2007; Jacob & Furgerson, 2012) also known as interviewing the participant based on the themes. The interview questions were arranged under broad areas identified as research questions in the **Chapter One** in order to ask every participant the same question.

The interviews started with the software developers and concluded with the software developer's managers. This enabled the researcher to clarify some questions which the developers were not comfortable to answer.

The researcher was an active participant throughout the process. The researcher used unstructured interviews, basically, the interviewer is interested in hearing from the interviewee

, so, the interviewee may be asked a variety of different open-ended questions. He was given access to source code and was actively involved in the set up of version control systems.

OBSERVATIONS

This is a method of intentionally noticing, transcribing and/ recording events that have been seen in the selected area of study (Runeson & Höst, 2009). According to Brinkerhoff, Ortega & Weitz (2013), observation is a way of conducting a research by participating, observing cases in the field and/or interviewing where necessary. The details of observation may be in the form of notes, images, videos or voice. Basically, there are two types of observations in data collection. These are participant and non-participant observations.

Participant observation is more appropriate in qualitative research since it gives the researcher the opportunity to collect data on a natural setting in places believed to be relevant to the research questions (Mack et al., 2011). This study used participant observation where the research took part in the phenomenon being studied (Palmer & Griggs, 2010; Gratton &

Jones, 2010) by acting as an employee of the participating organization. The researcher observed the steps taken to setup version control systems, the usage and various commands needed to run the software.

The researcher also made notes and made video tutorials as the team were setting up the software. The major drawback of this method is that the researcher was required to establish himself as the employee of the participating organization in order to avoid awkward behavior. This type of data gathering technique also requires the skill of a researcher to balance the role of being an observer and an employee (Neale, 2009). Participant observations, as further stated by Neale (2009), put participants at risk, especially when studying prohibited/illegal activities.

DOCUMENT REVIEW

Document review is a way of collecting data by going through existing documents concerning the study to find relevant information that can be used for data analysis (CDC, 2009). This method can be used to triangulate other methods being used in the study to improve data validity. This method was used to suggest questions for further inquiry during data collection. Documents included manuals and screenshots from the software manuals. The documents were sorted according to the major headings described in the interview protocol.

3.8 DATA ANALYSIS

Data analysis is the process of arranging data according to themes and patterns so that they can be transformed into the findings by linking these patterns/themes in the coding levels to literature or other theories (Boeije, 2009). This study used a structural coding approach (Saldana, 2009) which is one of the approaches used in qualitative research to find themes to describe a certain case. According to Saldana (2009), structural coding is used when the study uses many participants in qualitative research.

The goals of the data analysis are:

- To make some type of sense out of each data collection
- To look for patterns and relationships both within a collection, and also across collections, and
- To make general discoveries about the phenomena you are researching.

While data analysis in qualitative research can include statistical procedures, many times analysis becomes an ongoing iterative process where data is continuously collected and analysed almost simultaneously. Indeed, researchers generally analyse for patterns in observations through the entire data collection phase (Savenye, Robinson, 2004). The form of the analysis is determined by the specific qualitative approach taken (field study, ethnography content analysis, oral history, biography, unobtrusive research) and the form of the data (field notes, documents, audiotape, videotape).

An essential component of ensuring data integrity is the accurate and appropriate analysis of research findings. Improper statistical analyses distort scientific findings, mislead casual

readers (Shepard, 2002), and may negatively influence the public perception of research. Integrity issues are just as relevant to analysis of non-statistical data as well. In order to analyse data in this qualitative research, the researcher will use of atlasti, an analysis software package. Although some researchers suggest that disassembling, coding, and then sorting and sifting through the data, is the primary path to analysing data / data analysis. But as other rightly caution, intensive data coding, disassembly, sorting, and sifting, is neither the only way to analyse the data nor is it necessarily the most appropriate strategy. It has been argued that they also fit the notice, collect, and think process invariably also belonging to the data analysis process.

3.8.1SAMPLING STRATEGY

Sampling strategy is a laid down procedure that gives the researcher some guidelines to choose or select participants in a research. Sampling, as it relates to research is a process of determining how participants for a specific study should be selected (Onwuegbuzie & Leech, 2007). Qualitative research uses non-probability sampling as it is not intended to produce a numerical sample from the study (Wilmot, 2005).

A purposive sampling strategy was used in the study to collect data from a defined population of different employees in the development department at Multipay. This is supported by (Palinkas et al., 2013) who stated that purposive sampling is generally meant to select a small number of samples that will yield the most useful information about a particular case. The method was aimed at identifying participants who have knowledge of VCS. The criteria for choosing the participants was based on being a software developer, a software tester or a technical lead. Semi-structured interviews were carried out with the technical lead and testers while questionnaires were distributed to software developers and testers.

3.8.2 TARGET POPULATION

Target population is the group of elements that possesses varying characteristics that make them suitable for selection in the study area. (Fricker, 2006) refers it to a group of elements to which the researcher intends to make or draw conclusions from. The target population of this study comprised all the employees from the software development department.

3.8.3 SAMPLING POPULATION

A sample can be seen as the elements that represent the target population to be studied by the researcher (Phrasisombath, 2009). In a purposive sampling strategy, the researcher might not be able to choose a sample size in advance since there is no specified number of people

within the sample population that should be studied. According to Mack et al. (2011), purposive sample sizes are often determined on the basis of theoretical saturation. This means that the data collection process will terminate when participants seizes to provide additional insights to the research question. The participants were chosen according to the personal judgment of the researcher whereby only those who were perceived to have knowledge about the prepaid electricity meters were made to be part of this study.

3..8.4 TRIANGULATION

This is the term used to refer to the use of at least two different methods in studying a research issue (Oosthuizen, 2009). This method was meant to confirm the validity of data in order to reduce the level of uncertainty. In this study, two types of triangulation were used; data and method triangulation (Hussein, 2009). Data triangulation was meant to collect data from various levels of people since technical leads, developers and software testers might provide different data in the same study. Method triangulation was achieved by interviewing technical leads, developers and software testers. Data was also collected from the developers through questionnaires. Also, the researcher had the privilege to go through the documents of the participating organisation and do participant observation.

3.9 ETHICAL CONSIDERATIONS

Ethics refers to the appropriateness of the researcher's behavior in relation to the rights of the participants of the study (Saunders, Lewis & Thornhill, 2009). The researcher must strike the balance between the quest for information and the rights of the participants. During information gathering, participants must be made aware of how data will be collected, analysed and reported.

The sensitivity of source code and software applications is very high and need to be protected as it an intellectual property to the organisation. The researcher was obliged to make sure that participants will not be harmed by their participation in the study. During the interviews it was necessary to hide the identity of participants of the study to preserve confidentiality.

The statement to acknowledge the consent of the participants was read at the beginning of the interview. It was also attached to the questionnaire as the cover page. In data analysis, participants were identified using the codes. The participants had the right to refuse to answer certain interview questions.

3.10 CONCLUSION

This chapter described the research design and methodology that was followed in this study involving the qualitative research approach, design science methodology, case study and expert review. The sampling strategy, target population, sampling procedure and data collection instrument was also discussed. Ethical considerations have also been highlighted. The data were code according to various themes to facilitate easy comprehension and analysis. Now that data has been collected and analysed, the next chapter will present and discuss the findings that emerged from the coded data.

CHAPTER FOUR: DATA PRESENTATION AND DISCUSSION

4.1 INTRODUCTION

The previous chapter described the methodology that was employed to obtain data from relevant sources of this study. The chapter sufficed by highlighting how data was analysed to identify themes in order to facilitate data presentation. Now that data has been gathered and analysed, the focus of this chapter is to present and discuss the themes that were identified during data analysis.

4.2 PARTICIPANTS INVOLVED IN THIS RESEARCH.

The research was targeted to professionals in the software development department. In this department 6 software developers ,1 technical lead,2 database administrators ,2 system administrators,3 software testers and 1 project manager partook in the data collection exercise. This study made use of a mixed method approach which combined questionnaires and interviews to collect data.

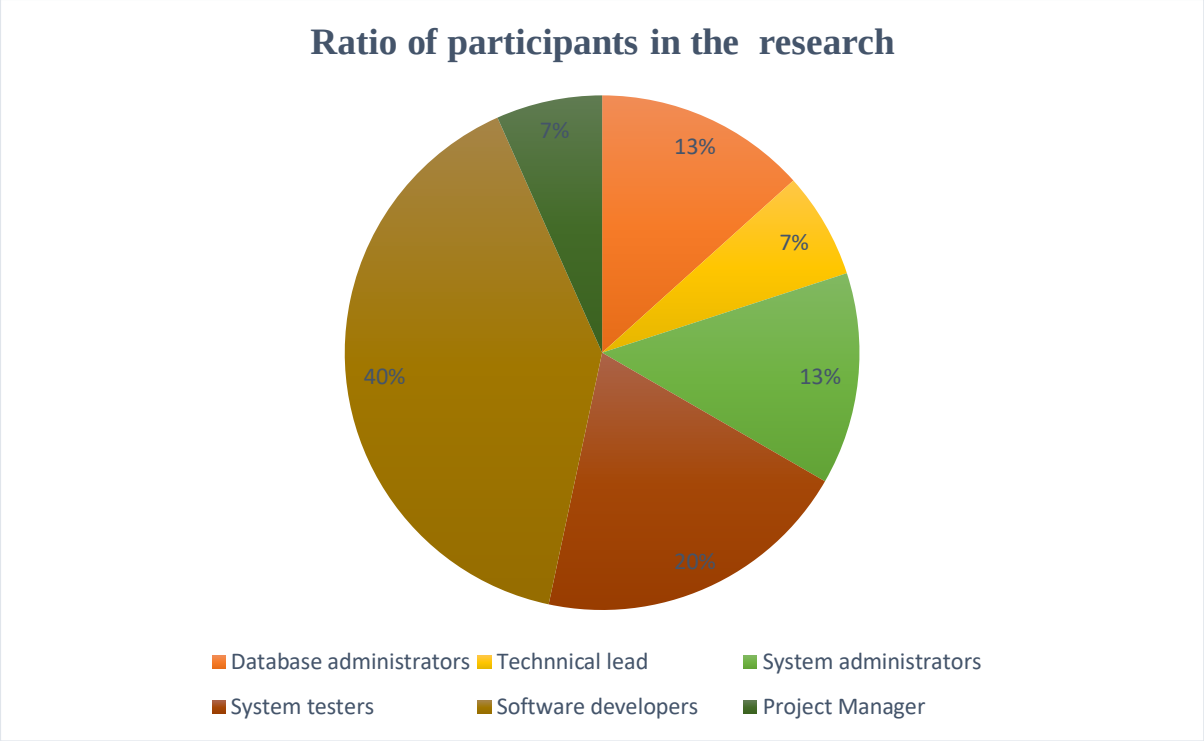

Figure 4.1: Participants involved in the study

COMPARISON BETWEEN RESPONDENTS AND NON-RESPONDENTS

Of the 16 participants that were involved in this research, 13 responded to the research instruments used in this study.86.7% represents 13 respondents and 13.3 % represents 2 participants that did not respond. This research defined not responding as either the participant failing to return questionnaires or failing to approve a request to carry out an interview with them.

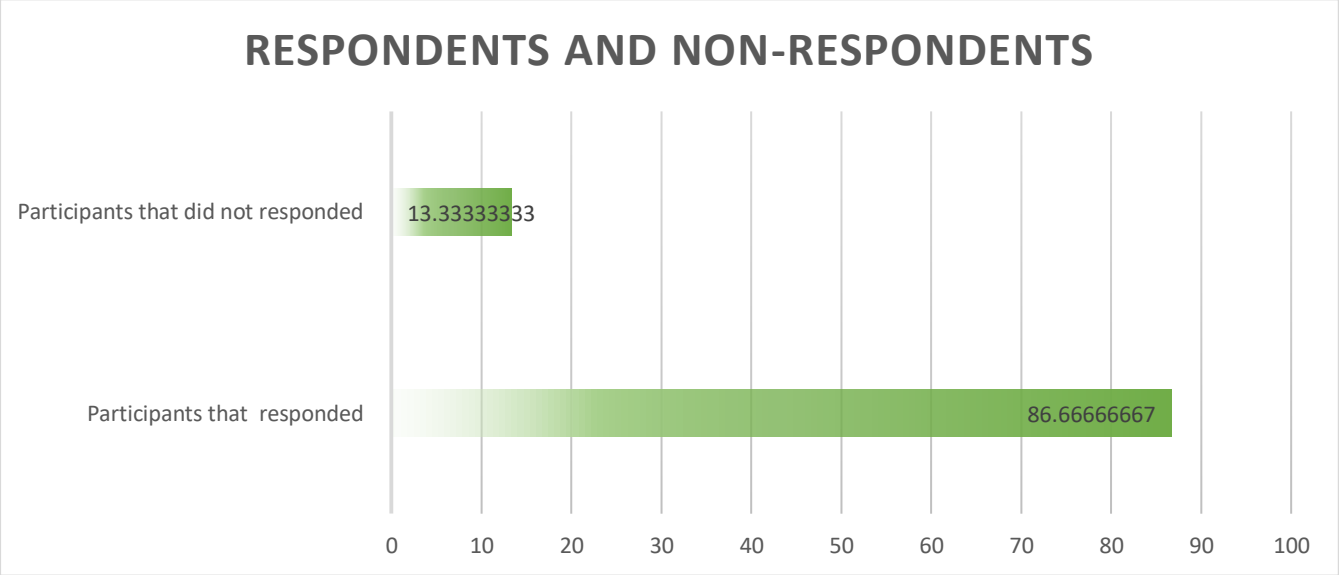

Figure 4.2: Comparison between respondents and non-respondents involved in the study

4.3 DATA PRESENTATION AND FINDINGS

The previous section introduced the purpose of this chapter. This section will provide an overview of how findings from qualitative data can be presented in a study.

The presentation of qualitative data is based on the narrative form. The findings in qualitative research should be grouped according to the themes developed using thoughtful verdicts (Roberts, 2004). The clarification of the theme(s) can be demonstrated by including the participants' quotes from data collection which help to bring the case of truth and believability (Pitney & Parke, 2009). Burnard et al. (2008) in their study on "Writing a qualitative research report" further elaborated that the key findings should be firstly reported under the main themes supported by participants' quotes as a means of illustration. Secondly, the findings are then discussed separately in relation to what was studied.

Before data can be presented, it must be analysed using the appropriate data analysis technique. In this study, structural coding (Saldana, 2009) was used to group the segment of data related to a specific research question using codes. The codes developed from data analysis are used to come up with themes that will facilitate data presentation. King & Horrocks (2010) stated that a theme should capture an important issue about a research question and should be based on the degree of repetition (recurring patterns). In contrary to this, (Guest, MacQueen, & Namey, 2012) suggested that the theme should be developed based on the number of participants who have mentioned it as opposed to repetition of the item in the data.

4.3.1 COLLABORATION AMONG DEVELOPERS

Terms	**No. of participants**
Scrum meetings	12
Passwords	8
Fetch	7
User names	13
Repositories	8
Internet link	9
Continuous integration	12
Testing	11
Workflow	3

Configuration	7
Push	7
Pull	7
Blame	7
Remote	7
Clean	2
Init	7
Clone	7

Table 4.1 Collaboration among developers

From the above codes, three themes were developed. These are used for access when developers are accessing remote repositories. These are user access controls (passwords, emails, user names, configuration), collaborations tools such as (scrum meetings, internet link, pull fetch, pull, blame, remote, clone) and integration controls such as (testing, blame, test scripts). These themes are presented in detail in section 4.3.

4.3.2 SOURCE CODE MANAGEMENT

Techniques	**No. of participants**
Release notes	12
Branching	7
Commits	8
Debugging	7
gamma test	3
Bugs	7
Bug tracker	4
Bisect	7
Git ignore	7
Merge	7
Status	7
Mv	7

Checkout	5
Describe	5
Grep	7
Test version	3

Two themes were formulated from the above codes. These are source code development (branching, commits, debugging, bugs, bisect, git ignore, merge and checkout) and test issues (gamma test, release notes, test version,)

4.4 FINDINGS

This section will present the findings from the study in the form of themes grouped according to research questions. These themes are the research findings of this study in relation to the research questions **(See chapter 1: section 1.4**). The themes were used to determine methods or tools that can be employed by the development department to come up with a seamless collaboration. (Bales R. F., Parsons T.,1953). The themes also aimed at looking at ways to effectively and manage source code and testing.

4.4.1 What is the best approach to effectively allow developers to collaborate in one project?

The research question aimed at establishing the different channels that the development team used to access the central repository. It also looked at collaboration methods that were available for the developers. The information was obtained through semi-structured interviews and document review from the system administrator, software developers and lead technical. There was high level of cooperation by the participants in providing necessary information. The system administrator presented documents for review, the researcher also had observation on the setup of VCSs.The department uses a distributed VCS, called **git**.

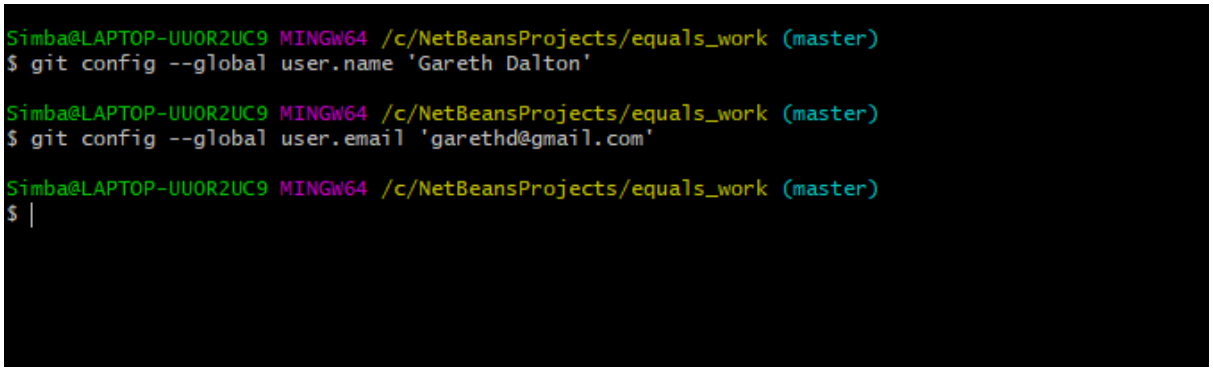

Figure 4.3: Adding users to a repository for identity.

The above figure shows a demonstration by a system administrator to set up a developer on their local repository. Usernames and emails are used to identify different developers working on a project. Once a user has been added all activity the user does is recorded in a log file against their name, the technical lead can use these user names to track changes. The System administrator sets up a Linux server as a central server/repository where all developers will fetch/put their development in after they are done developing.

On the server all users use the set username and password to access source code. The department currently only has one central repository. Each developer maintains her local repository but moves changes to the central repository after they complete their given tasks. The lead technical creates a shell/empty project and pushes it to the central repository so that its accessible by other developers. An illustration is shown below.

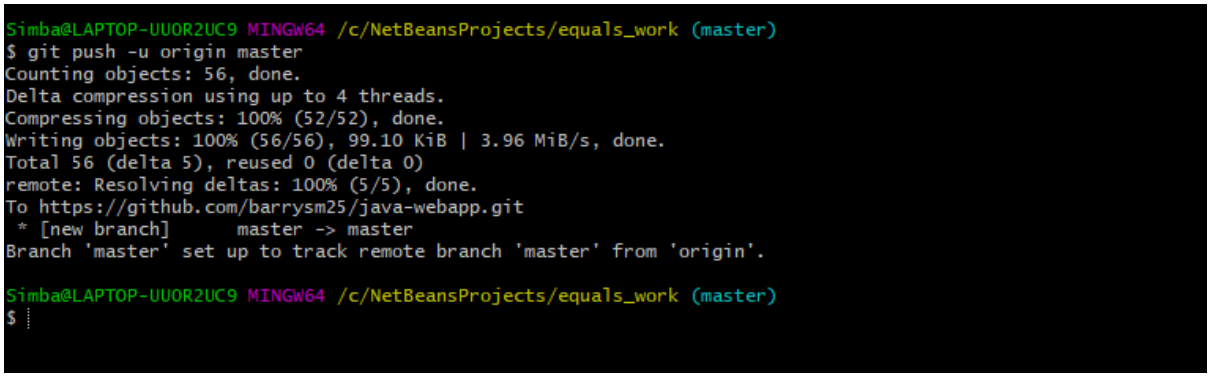

Figure 4.3: Lead developer pushes his local repositories to central repository.

After this has been done, other developers are given their tasks to complete, collaborating onto the central repository. To do this they need to remember their password which I

discovered they always forget and will need the system administrator to reset it almost 3 times per month. In order to start working on their given development tasks, a developer will need to clone the project as shown below.

```
git clone ssh://user@servername:port/path/to/project
```

Figure 4.3: Developer cloning from a central repository.

The above command will download source code on the remote repository to the local repository for a given user. Once they have the source code developers start work on their given development task.

In order to collaborate, they need to clearly understand what is expected of them, so team members will hold meeting, sometimes via skype or group chats, with the technical lead who will assign roles to each developer, system administrators are also involved in these meetings. A developer will have to pull /fetch an existing repo in order to get started and communicate with the technical lead once they are stuck and need help.

Before committing changes to the remote repository, a developer is required to pull but its not always the case as at times I observed the developers merely push without pulling first. The raised a lot of conflicts which could have been avoided if a proper framework had been put in place. In such scenarios the developer will need to use to use tools like git blame that shows who is to blame for what occurred. This drastically slows down development time as the team will need to solve this issue. In other instances, because the company relied on their hosting solution for the remote repository, there were challenges on connection in case the LAN was down and the developer who had the most up to date repository had to share it via email, flash disk or FTP.

4.4.2: How will version control allow concurrent project changes/edits by multiple team members?

This research question aimed at investigating the possible methods that are used by developers when working on the same project. The information was obtained through interviews and observation. The majority of this question was detailed in source code

management, which was entirely dependent on the software developers and technical lead. Although, the researcher managed to obtain data that was sufficiently enough to answer this research question, the majority of participants were not really aware of the majority of methods available in version control.

4.4.2.1 BRANCHING AND MERGING

The Git feature that really makes it stand apart from nearly every other SCM out there is its branching model. Git allows and encourages developers to have multiple local branches that can be entirely independent of each other. The creation, merging, and deletion of those lines of development takes seconds (Scott Chacon and Ben Straud,2017).

This means that developers can do things like:

Frictionless Context Switching. Create a branch to try out an idea, commit a few times, switch back to where you branched from, apply a patch, switch back to where you are experimenting, and merge it in.

Role-Based Code lines. Have a branch that always contains only what goes to production, another that you merge work into for testing, and several smaller ones for day to day work.

Feature Based Workflow. Create new branches for each new feature you're working on so you can seamlessly switch back and forth between them, then delete each branch when that feature gets merged into your main line.

Disposable Experimentation. Create a branch to experiment in, realize it's not going to work, and just delete it - abandoning the work—with nobody else ever seeing it (even if you've pushed other branches in the meantime).

Despite all these features available for developers to use, a few of these features are actually used. For instance, the software developers using are branching for experimentation, these are used in creating new codes that may not end up on the master branch**.** The researcher observed that a large number of developers actually create experimental branches on the central repository, this practise is widely used. Furthermore, the branch names are not meaningful such that a new developer may not know what functionality a particular branch as there no description is to support the creation of a branch. The situation is shown below

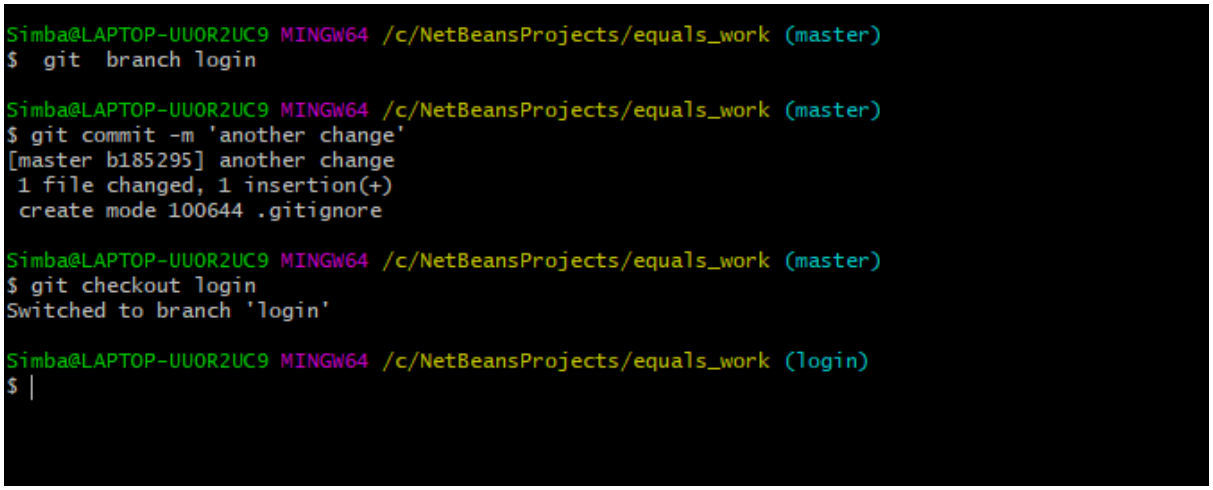

Figure 4.4: Creation of a new branch.

The screenshot above presents a situation where a developer is creating a branch called login. There is no description for the branch. After making the changes and experiments with the new branch, the developer commits changes, still with not enough information for other developers to gain an understanding of the commit. During the interview I asked the developers to why they used this practise and below are the answers.

Response 1:

"*As a developer I admit that I don't really much pay attention to branch names and commit messages, as at most times I am chasing deadlines. I just name a branch according to any name that comes to mind."*

Response 2:

"As a department we have not agreed to any naming conversions, that's why you see all these inconsistencies in naming of branches."

Response 3:

"For me writing code is an art. I am so good at writing English statements. I am not sure anyway on how comment on the commits"

As indicated by one of the respondents, the department has not agreed on any naming conventions, this has led to the cause of confusion on branch names.

MERGING

When a developer is satisfied that their new branch works as expected. This comes after alpha testing has passed and the developer will then need to merge source code on one branch to source code on the main branch. This is illustrated below,

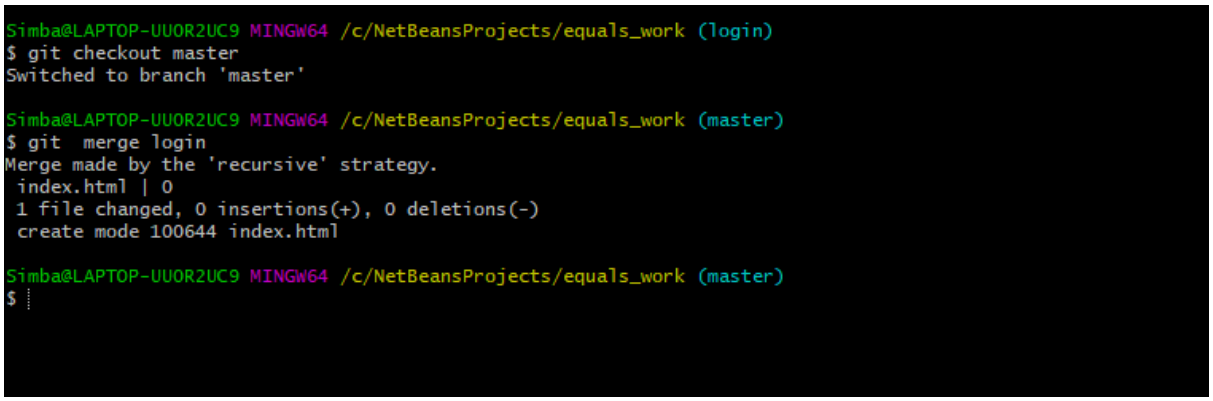

Figure 4.5: git merges

Git merge is a good tool when used correctly. A number of issues arise when developers merge an experimental branch without first pulling the remote main branch and testing that integrating the main branch with the experimental branch does not introduce new errors. The researcher also noted that a developer needs to commit changes while still on the experimental branch, and check out to the master branch, then merge the two branches. These steps must happen in sequence but I noted that, the developers do not always follow these steps which often leads to undesirable results.

4.4.2.2INSPECTION AND COMPARISON

After starting on a new project, the developer will create a lot of new files which could be java classes, html files and configuration files. These files will need to be added to the staging area or the index, this is an intermediate area where commits can be formatted and reviewed before completing the commit.

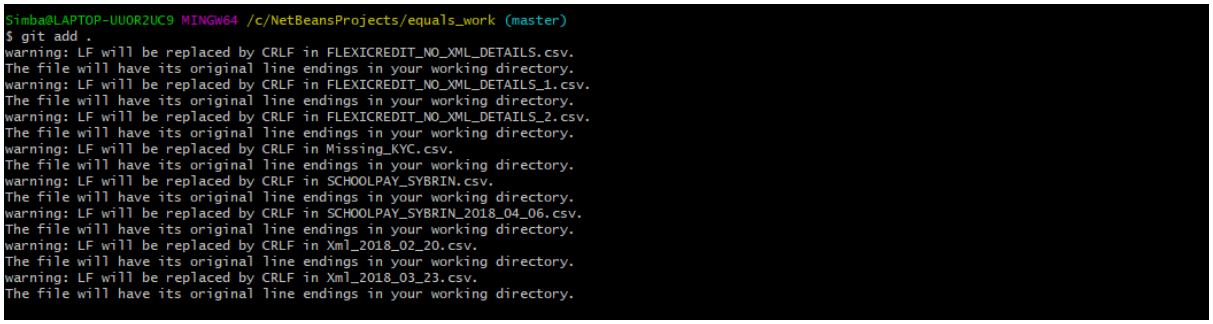

Figure 4.6: Adding files to the staging area

After the files are added to the staging area, they are ready to be committed, as illustrated below. When a developer checks the status of a project the VCS will show which files are ready to be committed and if a developer is satisfied they will then commit them to the repository.

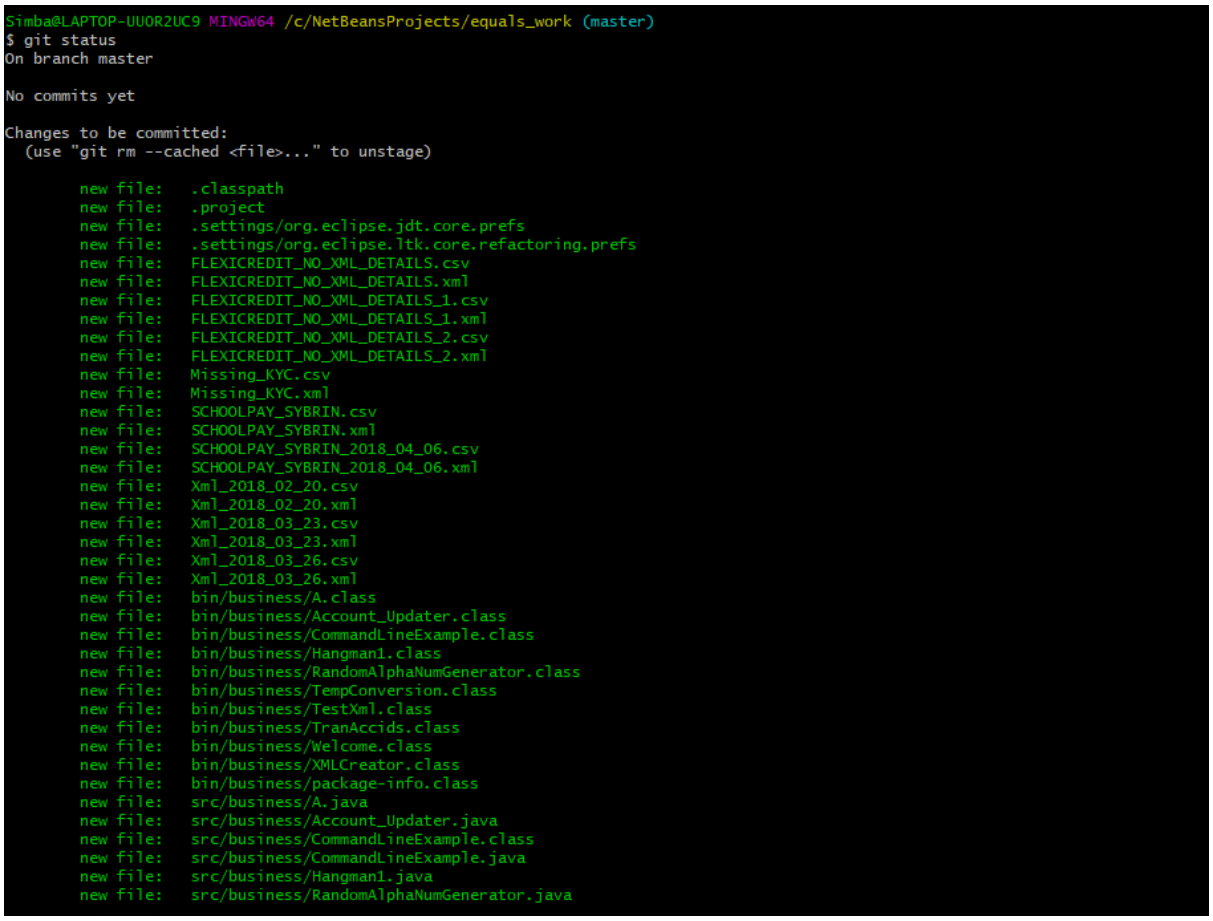

Figure 4.6: Adding files to the staging area

The researcher observed that the developers also ignored other files during development such that they were not committed. These files would be listed in a. gitignore file. This meant that the files listed in the file will not be included.

Figure 4.8: list of files to be ignored when committing.

Much of the log files were ignored when commits were being done. Above an error log file has been ignored for commitment.

An illustration is shown below. A file has not been added to the staging area and changes to this file will not be tracked.

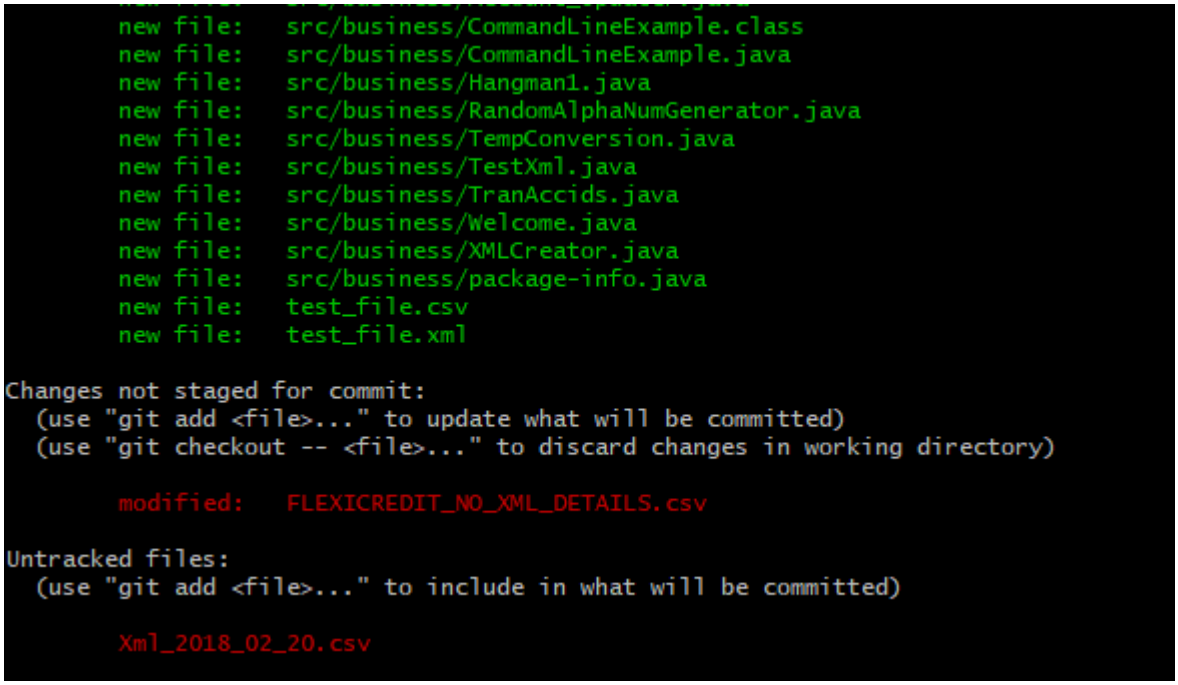

Figure 4.9: Untracked files will be listed in. git ignore

Developers also removed, renamed files from the staging area, these files will not be committed. To do this they used a simple rm command.

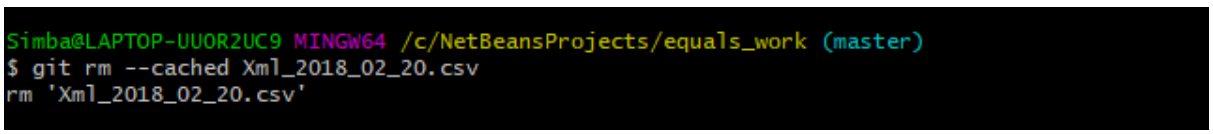

Figure 4.10: Removing a file from staging area

How do we manage a coordinated management to software releases given that more than one developer contributed to the project?

The research question aimed at establishing the best way possible to release a software product. Although the project manager was of the opinion that they were able to deal with this situation decisively, the software developers, technical lead, system administrator and testers raised concerns on the proper implementation of version control. In order for a software product to be released it goes through these rigorous steps, source code development, testing and release.

SOURCE CODE MANAGEMENT

Much of source code development issues have been highlighted previously. Developers need to work together on so many projects and meet to discuss the work flow to be adhered to. Once source code has been developed, it goes through alpha testing, its then integrated and tested again by the developers as unit tests. This is an iterative process and is done repeatedly until satisfied. (Tichy, 1985).When asked why integration takes time one developer responded as below:

"Source code works so well in separated units, but when integrated new errors emerges a lead I am supposed to create a working branch for the first phase of integration, this is not always the case as coordination is limited. Again, when branches are created we seldom forget the meaning and purpose of these branches."

TESTING

3 testers are employed in this department. Their main job is to test functionality of software as prescribed in the requirements specification. They get to know of the timelines of testing from the technical lead, who hands over release notes and from there, the testers create test scripts. The system administrator is also involved in this process as he will need to monitor performance by watching system logs and reporting any errors or warnings to the developers. The testers also keep test scripts of what they are testing and the results of each the researcher noted with great concern that the testers did not keep versions of their test scripts, but only recorded results. When asked why they do not version their test scripts they gave the following responses:

Response 1:

"Version control is only for software development, why should we use it? it needs a geek mind to understand how it works. I don't have time to learn"

Response 2:

"I am not well versed with VCS tools; my job is to test functionality and that's it. How does VCS work, will it increase my work load?"

The researcher observed that release notes were however kept in the VCS.

SOFTWARE RELEASE MANAGEMENT

The ultimate result of feature branches, distributed development, pull requests, and a stable community is a faster release cycle. These capabilities facilitate an agile workflow where developers are encouraged to share smaller changes more frequently. In turn, changes can get pushed down the deployment pipeline faster than the monolithic releases common with centralized version control systems(Spinellis, 2005).

The benefits of Git for product management is much the same as for marketing. More frequent releases mean more frequent customer feedback and faster updates in reaction to that feedback. Instead of waiting for the next release 8 weeks from now, developers can push a solution out to customers as quickly as the developers can write the code. Customer support and customer success often have a different take on updates than product managers. When a customer calls them up, they're usually experiencing some kind of problem. If that problem is caused by the company's software, a bug fix needs to be pushed out as soon as possible.

The project manager and technical lead need to manage software release such that, when an unforeseen bug is noticed by the customer they need to be heads up and know which build to look at. The researcher also noted that software release management needs to be crafted properly as one release of software can be developed for different consumers e.g. The department released an android application for CABS customers but also used the same

source code in that application for customers using Stanbic bank. When asked how the project lead managed to distinguish these two applications, his response was:

"I distinguish different product releases by the different folders the source code is placed in on the repository, each project has a unique folder structure. I also make notes about the release in the release notes."

4.5 DISCUSSION OF RESULTS

The previous section presented the methods employed by the development department for developers to collaborate on one project. It also highlighted how the team uses VCS to keep an auditable change history (e.g., what changed, when, and by whom) as well as practise in software release management.

Version control systems for quite some time present an integral part of development process and a must have tool for both individual developers and teams as well. Today, there is probably no serious developer or a company which doesn't use some sort of version control system on their daily basis. Indeed, these systems are relieving developers of tedious and error prone work of managing source code versions. Indeed, they bring to table a lot of features that now make us wonder how we managed to survive without them.

This section will provide a discussion of the findings obtained from the study.

4.5.1 COLLABORATION

The findings show that the development team has managed to implement, git, one of the most powerful version control systems. Its flexibility is reflected in the fact that it is used by both individual developers for small scale projects and huge companies such as Google, Facebook, Microsoft, Twitter, Eclipse for large projects. However, it also means that in order to use Git, both individuals and teams will have to invest significant time to master its possibilities and to use it to its full potential. Learning a syntax of few commands is not going to guarantee the team is doing version control properly. There is also multitude of written and unwritten patterns, anti-patterns, do's and don'ts, best practices, workflows, recommendations etc. From the interviews and observations make use of emails and usernames to identify each other on a project collaboration. This configuration but how ever not the best approach as I noticed most developers tend to forget their password or write them on pieces of paper which often poses as a security risk. A more reliable and secure way to collaborate will be

recommended in the next chapter. The is no consistence in the naming convections of usernames and password strength is not enforced. Developers use skype to communicate while collaborating and for small projects merging of small projects is not an issue but when it comes to bigger projects, more tools can be used which are not currently used at the moment. Version control system is also a great way to track contributions of individual team members, it makes it easier to find or at least narrow down bug sources, and it promotes code review and team member communication.

Although not its primary goal, this makes version control systems a great backup tool.

4.5.2 SOURCE CODE MANAGEMENT

Since source code files are basically textual files, version control systems can understand changes happening to particular file throughout various versions. This allows developers to rather easily roll back to any recorded point in the project history, or only inspect what the project or any file looked like at some point. A few team members have a good understanding of version control as they noted they have not invested enough time to learn the proper usage of the tool.

In the development team every team member besides working directory (working copy) has his own local repository to which he commits changes from working directory. Local repository allows member to privately work without disturbing other team members, or to be disturb by others. Most software developers have not mastered the art of naming branches and commit messages which adds more confusion when they come across a branch they worked on a while ago and do not realise what the functionality of the branch was.

The researcher also analysed the process of source code staging,

After a developer makes some changes in his working directory and wants to save them as a new version (snapshot) of his project, he prepares these changes by *adding* them to *Staging area* and *commits* them to Local repository. In order to make these changes available to another team member, he *push* them to Central repository. If other team members have done the same, we can *fetch* or *pull* their contribution from Central repository to his local repository.

Most of files such as log files are ignored when developers commit their changes, these will not contain any source code but only log messages, developers use for debugging.

From the investigation, the researcher was made to understand that although conflicts are not something to be feared (at least in Git), resolving conflicts can require careful analysis and discussion with other team members, which in the end can be quite time consuming. So, the department uses, one of the strategies in dealing with conflicts is to try to actually avoid conflicts. This means that one developer concentrates on much more frequent and smaller commits in order to avoid merging conflicts. Of course, that merely depends on the situation.

One of the most used methods by the team is to share the work each has done with their teammates. If commits are being done on a more frequent scale – conflicts will be easier to track and, in the end, to resolve. Conflicts are tied with the local machine on which the work is being done, so one shouldn't stop sharing on a more frequent basis due to being afraid of breaking the project. So, commit often and resolve conflicts as soon as you see them. Having frequent commits helps with resolving conflicts because in small scale commits there won't be as many conflicts as it would be in large scale commits.

4.5.3 TESTING

Constructing test script requires a considerable amount of work, including ongoing effort to cover new features and follow intentional code modifications. Testing is considered a best practice for software development in its own right, regardless of whether or not continuous integration is employed. The researcher was distressed to learn that testers do not use version control in their test scripts. Had this tool been used then keeping progress of tests was going to be very easy to track. Test scripts represent the health of any built, the fewer the bugs the healthier it is and bugs can be reported earlier using test scripts. A detailed set of steps to be conducted when testing using version control is brought forward in the next chapter.

4.5.4 DEPLOYMENT AND SOFTWARE RELEASE MANAGEMENT

The technical lead integrates work done by different developers into a single built for deployment. The integration process is manually done, a compiled source code is given an alias and stored in labelled directory to distinguish it from other applications. This process produces results but its not the best practise for version control. Proper recommendation for the best practice for software release management will be laid out in chapter 5, for the team leader to use.

4.6 CONCLUSION

This chapter presented and discussed various themes that were developed from the analysis of data. Version control systems for quite some time present an integral part of development process and a must have tool for both individual developers and teams as well. Today, there is probably no serious developer or a company which doesn't use some sort of version control system on their daily basis. Indeed, these systems are relieving developers of tedious and error prone work of managing source code versions. Indeed, they bring to table a lot of features that now make us wonder how we managed to survive without them. An analysis was made on the usage and best practise of version control from the previous chapter.

CHAPTER FIVE: CONCLUSION AND RECOMMENDATIONS

5.1 INTRODUCTION

The previous chapter presented and discussed the findings of the study in order to lay the foundation for developing the proposed framework. This chapter presents the conclusion of the study by re-stating the research problem and the objectives, with the proposed solutions presented as a framework. The recommendations that the development department need to adopt in order to effectively implement the best practise in version control when developing java web applications.

5.1.2 GOALS OF VERSION CONTROL SYSTEM

A version control system is a piece of software that helps the developers on a software team work together and also archives a complete history of their work. (Eric Sink, 2011)

There are three basic goals of a version control system (VCS):

- Development team to be able to work simultaneously, not serially.

Think of your team as a multi-threaded piece of software with each developer running in his own thread. The key to high performance in a multi-threaded system is to maximize concurrency. The goal is to never have a thread which is blocked on some other thread.

- When developers are working at the same time, their changes should not conflict with each other.

Multi-threaded programming requires great care on the part of the developer and special features such as critical sections, locks, and a test-and-set instruction on the CPU. Without these kinds of things, the threads would overwrite each other's data. A multi-threaded software team needs things too, so that developers can work without messing each other up. That is what the version control system provides.

- To archive every version of everything that has ever existed — ever. And who did it. And when. And why.

5.2 PROPOSED FRAMEWORK FOR THE BEST PRACTISES IN JAVA APPLICATIONS

The researcher has outlined the framework which should be adopted by the development team when they are developing java web applications with version control in mind. The framework has 4 pillars mainly collaboration, testing, software release management and source code management.

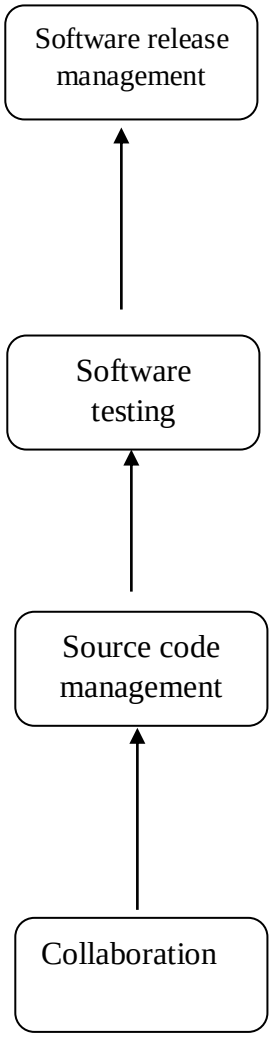

Figure 5.3 Pillars of the proposed framework

5.3.1 COLLABORATION

Coordination is a big concern in software development teams (Crowston et al.2005.) According to Curtis et al. (1988) large projects require more communication and coordination that can't be diminished by documentation. This requirement gets even bigger when the project team is distributed geographically or organizationally (Crowston et al. 2005).

When multiple developers are working on the same part of the project, it is important to use some kind of code management practices to handle all changes in the code effectively. Code management is significant challenge for project upkeep. In optimal distributed development environment, there are several capabilities that any project should have. Any project should be able to use available resources independently regardless of geographic location.

Project should also be able to plan practices and technology that supports the required coordination. Project should be capable to obtain shared understanding of requirements. Project should also be able to measure the "fit" of the software architecture. This means that the architecture of the project's software should be measured to be compatible with the rest of the system in the organization. Finally, project should be capable to react to changes and manage them properly. (Herbsleb, 2007)

Recommended practices for more efficient distributed software development are communication improving, more frequent deliveries that prevent mismatching, and making responsibilities in organization more transparent. (Herbsleb, 2007.)

(Herbsleb ,2007) One major challenge in distributed software development is managing the requirements. These requirements include all that specifies what a team should deliver. Integrating the necessary knowledge is challenging in distributed context. Communication has a key role in this kind of knowledge distribution, which cannot be completely relied on documentation.

In distributed software projects, people communicate with less people than in so called "inhouse" projects. Reasons for this are related with time zones, culture and language, and travelling difficulties. Less communication also means less effective communication. Loss of effectiveness is partially explained by delayed messaging frequency because of few overlapping work hours of communicating members. Participants in distributed projects often share relatively little context. They don't have much knowledge on what other project members are doing. This makes it difficult to initiate contact and can often lead to misunderstandings on the project. It can also make it difficult to stay up to date on the project

overall status. Different members can also use different kinds of tools, processes and practices, which can lead to issues in the project's progress. (Herbsleb, 2007).
Communication between different project members in distributed software developing is usually relied on "formal" channels, which are scheduled meetings or asynchronous channels like email, skype or documentation. These channels don't courage to impromptu conversations that would help to build relationships between members. These channels are also slow if members have only few overlapping work hours. (Damian, 2007. Herbsleb, 2007.)
One important form of collaboration especially in software development teams is knowledge collaboration. This means that development community members share knowledge among other members who have different roles such as leaders, developers and testers.

The setup helps members to gain all required knowledge. (Kakimoto et al.2006). GitHub is a code hosting site with collaborative and social features added on top of the Git version control system. As of 2014, it is the largest code hosting site in the world with over 10.6 million repositories. However, that number also includes personal projects and non-software repositories. (Kalliamvakou et al, 2014b). Developers have found that hosting their projects on GitHub has increased the amount of participation in their projects, citing the developer friendly interface and low barrier to entry as reasons for success (McDonald & Goggin's, 2013).

In addition to hosting projects in a private account on GitHub developers can make use of SSH keys for identification purpose when pushing/pulling source code from projects. This will offer great improvements to source code security as an SSH key is unique and can never be duplicated. The developer will not need to memorise their password as it will be stored in the SSH key.

5.3.2 SOURCE CODE MANAGEMENT

Source code management is another pillar for best practise when using version control. After developers collaborate, there is great need to properly manage the source code such conflicts are brought to a minimum. Below are the points for proper source management.

- Run diff just before every commit, every time

A developer must never commit their changes without giving them a quick review in some sort of diff tool.

- Read the diffs from other developers too

Every morning before a developer starts their own coding tasks, he should use his favourite diff tool to look at all the changes that everybody else checked in the day before. When a developer reads the diffs, two good things might happen:

The code might get better. Reading the diffs is like an informal code review. You might find some-thing that needs to be fixed and you might learn something. Maybe one of the co-workers is using a technique you don't know about. Or maybe reading the diffs simply gives you a deeper understanding of the project you are working on.

- Keep repositories as small as possible

And no smaller.

Since the DVCS model involves every developer keeping a complete copy of the repository on her desktop machine, it is best to be intentional about how much stuff goes into a single repository. It is not a good idea for a large corporation to have just one repository into which all projects go. If somebody else is also working on your branch at the same time, always do a pull before doing a push so that you can be sure to incorporate any changes they may have pushed before you. Try to avoid the above situation as much as possible. At least avoid working on the same files if you can't avoid working on the same branch.

- Group commits logically

Each changeset you commit to the repository should correspond to one task. A "task" might be a bug-fix or a feature. Include all of the repository changes which were necessary to complete that task and nothing else. Developers must avoid fixing multiple unrelated bugs in a single changeset.

- Explain the commits completely

Every version control tool provides a way to include a log message (a comment) when committing changes to the repository. This comment is important. If we consistently use good comments when we commit, our repository's history contains not only every change we have ever made, but it also contains an explanation of why those changes happened. These kinds of records can be invaluable later as we forget things. The researcher believes developers should be encouraged to enter log messages which are as long as necessary to explain what is going on. Don't just type "minor change". Tell the team what the minor change was. Don't just tell us "fixed bug 1234". Tell us what bug 1234 is and tell us a little bit about the changes that were necessary to fix it.

- Only store the canonical stuff

The best practice is to store everything which is created manually, and nothing else. I call this "the canonical stuff". Do not store any file which is automatically generated. A developer should store hand-edited source code. Developers should not store EXEs and DLLs. If developers use a code generation tool, they should store the input file, not the generated code file. If the team generates the product documentation in several different formats, they should store the original format, the one that they manually edit. If there are two files, one of which is automatically generated from the other, then they don't need to store both of them. They would in effect be managing two expressions of the same thing. If one of them gets out of sync with the other, then you have a problem.

- Don't break the tree

The benefit of working copies is mostly lost if the contents of the repository become "broken". At all times, the contents of the repository should be in a state which allows everyone on the team to continue working. If a developer checks in some code which won't build or won't pass the test suite, the entire team grinds to a halt. Many teams have some sort of a social penalty which is applied to developers who break the tree. For example, require the guilty party to put a dollar in a glass jar. (Use the money to take the team to go see a movie after the product is shipped.) Another idea is to require the guilty individual to make the coffee every morning. The point is to make the developer feel somewhat embarrassed, but not punished.

Anyway, the central repository is a place the team shares with the others on the development team. Respect them by being careful about what you push there. At a minimum, make sure that stuff builds on your machine before you commit and push. If a developer has an automated test suite, run it and make sure you didn't break anything.

- Use tags

Tags are cheap. They don't consume a lot of resources. The version control tool won't slow down if you use lots of them. Having more tags does not increase the team's responsibilities. So, tags cab be used as often. The following situations are examples of when you might want to use a tag:

When developers make a release, apply a tag to the version from which that release was built. A release is the most obvious time to apply a tag. When you release a version of your application to customers, it can be very important to later know exactly which version of the code was released.

Sometimes it is necessary to make a change which is widespread or fundamental. Before destabilizing your code, you may want to apply a tag so you can easily find the version just before things started getting messed up. Some automated build systems apply a tag every time a build is done. The usual approach is to first apply the tag and then do a "get by tag" operation to retrieve the code to be used for the build. Using one of these tools can result in an awful lot of tags, but I still like the idea. It eliminates the guesswork of trying to figure out exactly which code was in the build.

- Always review the merge before every commit.

Successfully using the branching and merging features of your source control tool is first a matter of attitude on the part of the developer. Create a new branch for every specific task, feature and bugfix and avoid having one big generic branch. Creating sub-branches is also useful, and it's often a good idea to create your own master development branch (named dev-1.2-description) and branch off that instead of master... just don't ever work directly on the main master branch. No matter how much help the version control tool provides, it is not as smart as you are. Developers are responsible for doing the merge. Think of the tool as a tool, not as a consultant. After your version control tool has done whatever it can do, it's your turn to finish the job. Any conflicts need to be resolved. Make sure the code still builds. Run the unit tests to make sure everything still works. Use a diff tool to review the changes. Merging branches should always take place in a working copy. Your version control tool should give

you a chance to do these checks before you commit the final results of a merge branches operation.

- Don’t comment out code

When using a VCS, developers shouldn’t comment out a big section of source code simply because they think you might need it someday. Just delete it. The previous version of the file is still in your version control history, so you can always get it back if and when you need it. This practice is particularly important for web developers, where the commented-out stuff may adversely affect your page load times.

- Build and test your code after every commit

Set up an automated build system which is triggered every time there is a new changeset in the repository instance on your central server. That system should build and test the code, broadcasting a report of the results to the entire team.

5.3.3 SOURCE CODE RELEASE MANAGEMENT AND WORKFLOW

A software product is released when satisfactory testing has passed. This means source code will have proceeded form alpha, beta and gamma testing. A gamma branch can be created when working on a new component or feature or bugfix. When ready to pilot your new component or feature or bugfix, merge your working branch to gamma and push it to the remote repository. Optionally make a beta branch and work off that if a developer feels he needs to test bits and pieces separately before giving the feature a complete trial run.

Code review

When satisfied, developers are supposed to inform the technical lead for him to do a code review. The technical lead will pull the gamma branch, do a diff and perform code review, placing comments in the code. The technical lead will then push back the gamma branch, so the developer will need to do a pull, to asses the comments placed by the technical lead. The developers are supposed to make corrections as requested, deleting my comments as they go along. When the technical lead has passed source code review with flying colours, the developers will merge gamma to master and push it. After this , the developers can start making a release. Make a release file name should be:

project_name.major_version.minor_version.build_number.war, e.g. cabszipit.1.1.0.war

Release Management workflow.

Below are 2 workflows a technical lead can adapt to integrate source code from different developers when making software releases.

Integration Manager Workflow

Another common Git workflow involves an integration manager — a single person who commits to the 'blessed' repository. A number of developers then clone from that repository, push to their own independent repositories, and ask the integrator to pull in their changes. This is the type of development model often seen with open source or GitHub repositories.

(Spinellis(, 2012).With this workflow , the technical lead will push the final release software out to consumers.

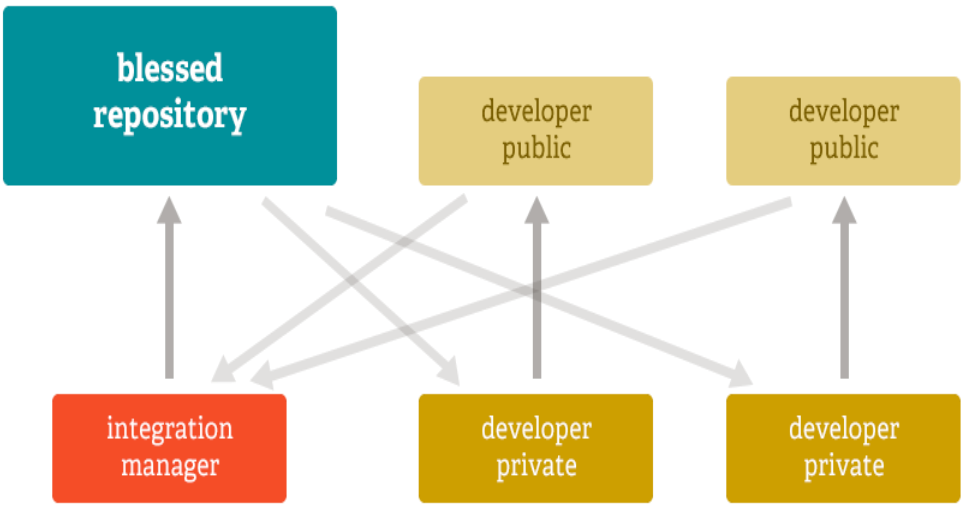

Figure 5.1 Integration Manager Workflow

Dictator and Lieutenants Workflow

(Spinellis(, 2012), For more massive projects, a development workflow like that of the Linux kernel is often effective. In this model, some people ('lieutenants') are in charge of a specific subsystem of the project and they merge in all changes related to that subsystem. Another integrator (the 'dictator') can pull changes from only his/her lieutenants and then push to the 'blessed' repository that everyone then clones from again.

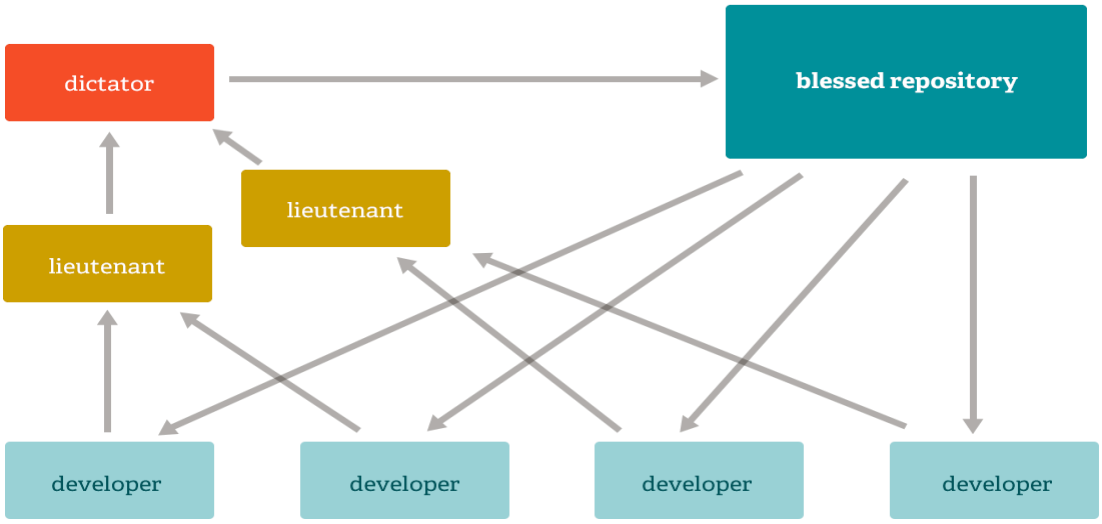

Figure 5.2: Dictator and Lieutenants Workflow

5.3.3 TESTING

Before the technical lead releases an application, it undergoes a thorough testing process to ensure that the software is working in the manner in which it was intended. There are four main stages of testing that need to be completed before a program can be cleared for use: unit testing, integration testing, system testing, and acceptance testing. If a build passes any prior test it is pushed to the next level and so on, if it fails results are sent back to the developers who makes changes and push back the build to be tested. This is shown below. The builds move through a conveyer belt of tests.

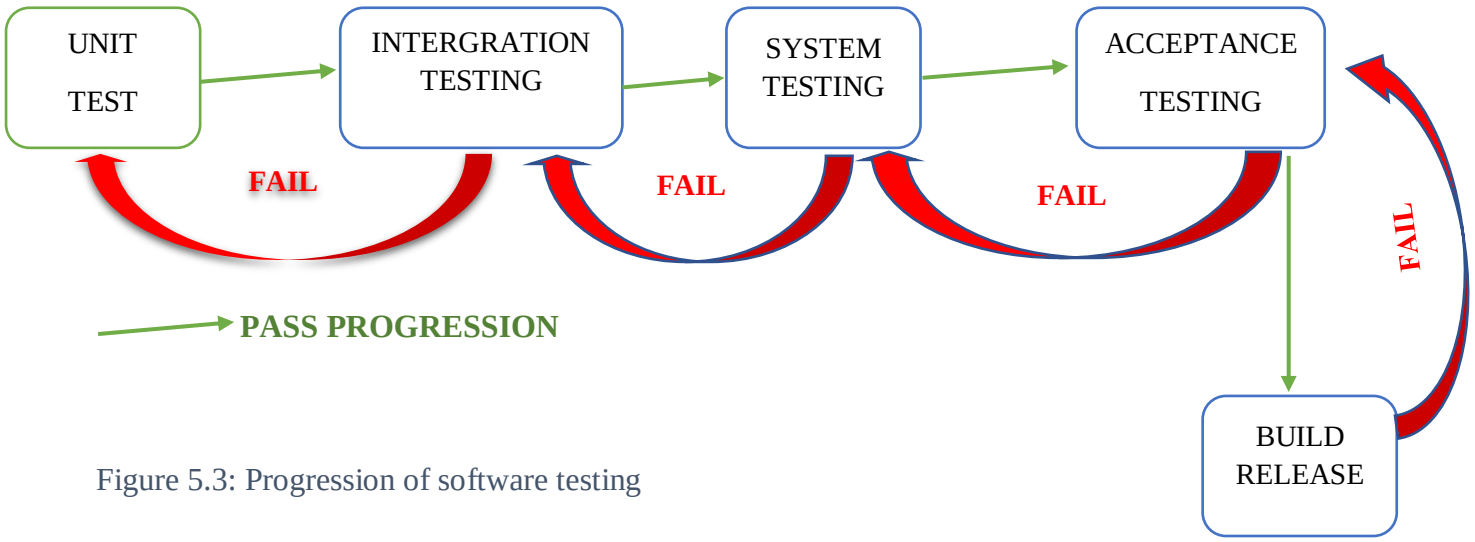

Figure 5.3: Progression of software testing

All these tests have results that need to be captured and stored in versions. The best practise for software testers is to match each test script with the build number of the software version

they are testing. Test scripts are merely text-based documents that testers input notes, results and comments. These too need to be versioned for future reference.

5.4 EVALUATION OF THE PROPOSED FRAMEWORK

The previous section described and presented the proposed framework. This section will discuss the evaluation procedure.

As an evaluation step in the development process, expert reviewers have been presented with the initial framework to assess that it harmoniously portrays every attribute that are needed to accomplish specific tasks. These attributes ensure that the proposed framework has the quality of being near to the true value of the artefact so that it can fit to the organisation and serve the purpose. Here, the purpose is to present the best practise in version control when developing java web applications. The comments from the expert reviewers will be used to refine the framework before the final presentation. Justification will also be made in the comments not considered for validation of the framework.

5.5 CONTRIBUTIONS TO PREVIOUS STUDIES

The previous section has provided the validation of the proposed framework. This section will present the contributions of the proposed framework to both the operating environment and the knowledge base.

As stated by Hevner et al. (2004), an information system research carried out under the design science paradigm should add value to the knowledge base and the application domain. This is done by extending theories and methods that have been presented by previous studies. The key contribution of this study is a framework for version control in java web applications development. The framework can be used by software developers and can be applied to all the stages in a software project life cycle. This framework will give development team knowledge of what's the expected behaviour is for that code, since a version of everything is recorded. Using this framework testers have an ability to review and accept tests using standard code-review processes, Code review saves time, reduces bugs and helps keep the team on the same page. Those benefits translate to writing tests just as well as they do for writing code. "When you're writing tests for four hours a day you really get in the zone and can easily get

5.6 RECOMMENDATIONS

The previous section demonstrated how the proposed framework will contribute to the existing work that has been done by other scholars. This section will present the necessary recommendations to the power utility.

The development team should invest in significance amount of time, nurturing new developers on how to use version control, version control is easy to implement but as the researcher found out, it becomes tricky to implement when the team is pressed with deadlines, and ever-changing user requirements when developing software projects.

5.7 LIMITATIONS OF THE STUDY

This section will present the limitations of this study so as to come up with the direction for future research.

The main limitation of the proposed framework lies upon the scope of the study, the design science paradigm and the findings. The study only focused on the development team, other important stakeholders could not be included in this research such as customers, project sponsors, and the authors who will write manuals for the software that is being developed. Another limitation was access to the actual source code as there are licence agreements between the company and the sponsor such that the source code will be owned by the sponsor and revealing it to another party will be considered a breach. The researcher would have wanted to dive deeper into software testing but, testing takes a long time and testers are much of data quality assurance which on its own requires tremendous skills.

5.8 FUTURE RESEARCH

This section will present directions/ guidelines for future work which can be used as a starting point for providing improvements on the proposed framework. These guidelines are underpinned on the limitations of the study discussed in the previous section.

Future work in this study may commence by considering participants who were difficult to reach in this study. The research could also be a shell for other frameworks who are targeting open source application development, as it will comprise of teams dispersed across a wide geographical region. It can also be customised for other programming languages.

5.9 CONCLUSION

The major research question of this study was answered by developing a framework for the best practise in version control when developing java web applications. The proposed framework is expected to be a guide and a reference for software development teams pursuing java web development's proposed framework was validated through expert review. Therefore, it can be concluded that it will meet the organisational needs and will fit well within the infrastructure to be implemented.

REFERENCES

Guba, E. G., & Lincoln, Y. S. (1982). Epistemological and methodological bases of naturalistic inquiry. ECTJ, 30(4), 233–252.

Holden, M. T., & Lynch, P. (2004). Choosing the Appropriate Methodology: Understanding Research Philosophy. The Marketing Review, 4(4), 347–409.

Kothari, C. R. (2012). Research Methodology: An introduction. In Research Methodology: Methods and Techniques (p. 418).

Rowlands, B. (2005). Grounded in Practice: Using Interpretive Research to Build Theory. The Electronic Journal of Business Research Methodology, 3(1), 81–92.

Saunders, M., Lewis, P., & Thornhill, A. (2009). Research Methods for Business Students (5th ed.). Essex, England: Pearson Education Limited.

Sobh, R., & Perry, C. (2006). Research design and data analysis in realism research. European Journal of Marketing, 40(11/12), 1194–1209

Riemer, F . J., Lapan, S. D. & Quartaroli, M. T. (2012) Qualitative research: an introduction to methods and designs San Francisco: Jossey -Bass.

May, T. (2011)Social research: issues, methods and process 4th ed. Maidenhead: Open University Press.

Matthews, B. & Ross, L. (2010) Research methods: a practical guide for the social sciences Harlow: Longman.

Marczyk, G. R. , DeMatteo, D. & Festinger, D. (2005) Essentials of research design and methodology . Hoboken: Wiley.

Jupp, V. (2006) The Sage dictionary of social research methods. London: Sage.

Banister, P., Bunn, G., Burman, E., & Daniels, J. (2011). Qualitative Methods In Psychology: A Research Guide. London: McGraw-Hsill International.

Beiske, B. (2007). Research Methods: Uses and limitations of questionnaires, interviews and case studies, Munich: GRIN Verlag.

Bryman, A. (2012). Social research methods (5th ed.). Oxford: Oxford University Press.

Bryman, A., & Allen, T. (2011). Education Research Methods. Oxford: Oxford University Press.

Bryman, A., & Bell, E. (2011). Business Research Methods (3rd ed.) Oxford: Oxford University Press.

Feilzer, M. Y. (2010). Doing mixed methods research pragmatically: Implications for the rediscovery of pragmatism as a research paradigm. Journal of Mixed Methods Research, 4(1), pp.6-16.

Flick, U. (2011). Introducing research methodology: A beginner's guide to doing a research project. London: Sage.

Goddard, W. & Melville, S. (2004). Research Methodology: An Introduction, (2nd ed.) Oxford: Blackwell Publishing.

Gulati, P. M. (2009). Research Management: Fundamental and Applied Research, New Delhi: Global India Productions.

Institut Numerique, (2012). Research Methodology, http://www.institut-numerique.org/chapter-3-research-methodology-4ffbd6e5e3391 [retrieved 3rd October, 2014].

Kothari, C. R. (2004). Research methodology: methods and techniques. New Delhi: New Age International.

May, T. (2011). Social research: Issues, methods and research. London: McGraw-Hill International.

Monette, D.R., Sullivan, T. J., & DeJong, C. R. (2005). Applied Social Research: A Tool for the Human Services, (6th ed.), London: Brooks Publishing.

Neuman, W. L. (2003). Social Research Methods: Qualitative and Quantitative Approaches, London: Allyn & Bacon.

Newman, I. (1998). Qualitative-quantitative research methodology: Exploring the interactive continuum. Carbondale: Southern Illinois University Press.

Ã–stlund, U., Kidd, L., WengstrÃ¶m, Y., & Rowa-Dewar, N. (2011). Combining qualitative and quantitative research within mixed method research designs: a methodological review. International Journal of Nursing Studies, 48(3), pp. 369-383.

Podsakoff, P. M., MacKenzie, S. B., & Podsakoff, N. P. (2012). Sources of method bias in social science research and recommendations on how to control it. Annual Review of Psychology, 63, pp.539-569.

Rowley, J. (2012). Conducting research interviews. Management Research Review, 35(3), pp.260-271.

Saunders, M., Lewis, P., & Thornhill, A. (2007). Research Methods for Business Students, (6th ed.) London: Pearson.

Silverman, D. (2013). Doing Qualitative Research: A practical handbook. London: Sage.

Snieder R. & Larner, K. (2009). The Art of Being a Scientist: A Guide for Graduate Students and their Mentors, Cambridge: Cambridge University PressCrowston, K., Wei, K., Li, Q., Eseryel, U. Y., & Howison, J. (2005). Coordination of free/libre and

open source software development.
Curtis, B., Krasner, H., & Iscoe, N. (1988). A field study of the software design process for large
systems. Communications of the ACM , 31 (11), 12681287.
Dabbish, L., Stuart, C., Tsay, J., & Herbsleb, J. (2012). Social coding in GitHub: Transparency
and collaboration in an open software repository. Paper presented at the Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW, 12771286
Damian, D. (2007). Stakeholders in global requirements engineering: Lessons learned from practice. Software, IEEE , 24 (2), 2127.
Gousios, G., Pinzger, M., & Deursen, A. V. (2014). An exploratory study of the pullbased software development model. In Proceedings of the 36th International Conference on Software
Engineering (pp. 345355).
ACM.
Gurbani, V. K., Garvert, A., & Herbsleb, J. D. (2005, May). A case study of open source tools
and practices in a commercial setting. In ACM SIGSOFT Software Engineering Notes (Vol. 30,
No. 4, pp. 16).
ACM.
Herbsleb, J. D. (2007, May). Global software engineering: The future of sociotechnical coordination. In 2007 Future of Software Engineering (pp. 188198).
IEEE Computer Society.
Hu, D., & Zhao, J. L. (2008). A comparison of evaluation networks and collaboration networks in
open source software communities. AMCIS 2008 Proceedings , 277.
Kakimoto, T., Kamei, Y., Ohira, M., & Matsumoto, K. (2006, September). Social network analysis on communications for knowledge collaboration in oss communities. In Proceedings of
the International Workshop on Supporting Knowledge Collaboration in Software Development
(KCSD'06) (pp. 3541).
Kalliamvakou, E., Damian, D., Singer, L., & German, D. M. (2014). The codecentric collaboration perspective: Evidence from github . Technical Report DCS352IR, University of
Victoria.
Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., & Damian, D. (2014b).
The promises and perils of mining GitHub. In Proceedings of the 11th Working Conference on
Mining Software Repositories (pp. 92101).
ACM.
Lanubile, F., Ebert, C., Prikladnicki, R., & Vizcaíno, A. (2010). Collaboration tools for global
software engineering. IEEE software , (2), 5255.

McDonald, N., & Goggins, S. (2013, April). Performance and participation in open source software on github. In CHI'13 Extended Abstracts on Human Factors in Computing Systems (pp. 139144).
ACM.
Robbins, J. E. (2002). Adopting OSS methods by adopting OSS tools. CollabNet, Inc .
Scacchi, W. (2002, February). Understanding the requirements for developing open source software systems. In Software, IEE Proceedings(

Stuart Yeates, “What is version control? Why is it important for due diligence?“, January 2005

Michael Ernst, “Version Control Concepts and Best Practices“, September 2012

Ilya Olevsky, “Why version control is critical to your success?“, March 2013

Martin Fowler, “Version Control Tools“, February 2010

Chris Nagele, “An introduction to version control“

https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Tools/Source_Control

https://stackoverflow.com/questions/1408450/why-should-i-use-version-control

"Rapid Subversion Adoption Validates Enterprise Readiness and Challenges Traditional Software Configuration Management Leaders". Collabnet. May 15, 2007. Retrieved October 27, 2010. "Version management is essential to software development and is considered the most critical component of any development environment."

Wheeler, David. "Comments on Open Source Software / Free Software (OSS/FS) Software Configuration Management (SCM) Systems". Retrieved May 8, 2007.

O'Sullivan, Bryan. "Distributed revision control with Mercurial". Retrieved July 13, 2007.

Collins-Sussman, Ben; Fitzpatrick, B.W. and Pilato, C.M. (2004). Version Control with Subversion. O'Reilly. ISBN 0-596-00448-6.

Wingerd, Laura (2005). Practical Perforce. O'Reilly. ISBN 0-596-10185-6.

Collins-Sussman, Ben; Brian W. Fitpatrick, and C. Michael Pilato. "Version Control with Subversion". Retrieved 8 June 2010. "The G stands for merGed, which means that the file had local changes to begin with, but the changes coming from the repository didn't overlap with the local changes."

Accurev Concepts Manual, Version 4.7. Accurev, Inc.. July, 2008.

APPENDIX A: INTERVIEW GUIDE

My name is Simbarashe Makwangudze, I am pursuing MSc. Information Systems Management (MISM) with MSU. I am doing a research on a framework for the best practise in Developing Java Web Applications with Version Control. The study is aimed at developing a framework for the best practise in software development. For this purpose, semi-structured interviews will be conducted with key participants from Multipay Solutions software development department. You have been chosen to participate in this study because of your position and your duties in the power utility.

During the interview, I would like to discuss the following topics: collaboration of software developers working from different locations, source code security mechanisms, software testing using version control and challenges in software release management. With these topics in mind, I will be able to come up with a framework of the best practise in using version control in software development particularly in java web applications.

Interviews will last for about 45 minutes and questions will deal with version control in software development. The information obtained will be used solely for the purposes defined by the study. At any time, you can refuse to answer certain questions, discuss certain topics or even put an end to the interview without prejudice to yourself. To facilitate the interviewer's job, the interview will be recorded if you grant me the permission. However, the recording will be destroyed as soon as it has been transcribed. All interview data will be handled so as to protect confidentiality. Therefore, no names will be mentioned and the information will be coded. All data will be destroyed at the end of data analysis.

Do you have any questions or anything that you would want it to be clarified before I can start the interview?

1. **What approach is used to effectively allow developers to collaborate in one project even though some may of whom may be geographically dispersed, making sure that only authentic and authorized developers will have access to source code.**

    a. What method is used for authentication and identity.
    b. How do team members communicate when working together on one project?

c. How do you maintain an audit trail of developer's activity?

2. **The developers will need to inspect and compare files, work on bug fixes and hot fixes, what is the best approach to this.**
   a. How do they distribute tasks?
   b. How do they resolve conflicts when working on one file?
   c. Who leads conflict resolution and merging of work done.

3. **How can software testers make use of software version control software when testing new software versions, approving software version, maintaining automated tests, creating release notes and code review**
   a. Do you currently use version control in testing?
   b. How do you maintain match between a source code version and a test script?

4. **How do we keep an auditable change history (e.g., what changed, when, and by whom?**
   a. How do you maintain sanity in source code?
   b. How do you merge each developer s work?
   c. How is source code release management handled?
   d. How do you handle branching?

APPENDIX B: SAMPLE QUESTIONNAIRE

Questionnaire for developers and testers

I am a post graduate student at the Midlands State University (MSU) undertaking a research. I am doing a research on "**a framework for the best practise in Developing Java Web Applications with Version Control**". The study is aimed at developing a framework for the best practise in software development., I am kindly inviting you to participate in this study by completing the attached questionnaires. It will require approximately fifteen minutes to be completed. There is no compensation for responding.

All information gathered will be entirely for academic purposes and no part will be used otherwise. Confidentiality will be highly preserved. The validity of the results will depend on obtaining a high response rate, hence your participation is crucial to the success of this study and please answer all the questions as honest as possible. Completed questionnaire should be returned by the 5th of May 2018.

Please do not write your name anywhere on this paper.

INSTRUCTIONS FOR ANSWERING THE QUESTIONNAIRE

Please **tick** the appropriate box where answers are provided and **fill** in the spaces provided.

1. Do you know version control? Yes ☐ No ☐
2. What is your level of knowledge of the use of version control?

Advanced	
Moderate	
Above average	
Average	

3. Which VCS are you familiar with?

Distributed ☐ Central ☐

4. Which major arrears do you find issues when using VCS?

development ☐ Testing ☐ Software release ☐

5. Do you spend time learning more about version control?

6. Yes ☐ No ☐

7. Can you briefly explain the steps you take when a software product is being launched?

---

8. Its known conflicts are expected when merging source code, how do you resolve them?

---

9. Do you coordinate test scripts with different software version?

Yes ☐ No ☐

10. What will you be looking for when testing application before its released?

---

11. What is your level of qualification?

Certificate
Diploma
National diploma
Higher national diploma

Degree
Masters
None

12. In your opinion, how can we best implement the best practise of version control?

---

13. After our discussion can you see the benefits of using version control?

END OF QUESTIONNAIRE, THANK YOU FOR YOUR PARTICIPATION